KICK Pascal V2.0

Das integrierte Pascal-Entwicklungssystem

MAXON
computer gmbh

KICK-PASCAL 2.0

Autor: Jens Gelhar

Lizenzbestimmung

Mit dem Erwerb dieser Software erhalten Sie das Recht zur Installation und Nutzung von KICK-Pascal 2.0 an einem Rechnersystem. Mehrplatzlizenzen auf Anfrage. Für die mit KICK-Pascal 2.0 erstellten kommerziellen und privaten Programme entstehen Ihnen keine Lizenzzahlungen an MAXON Computer. In der Infobox des jeweiligen Programmes, bzw. im dazugehörenden Handbuch, muß jedoch folgender Satz vermerkt sein: "**Erstellt mit KICK-Pascal 2.0 von MAXON Computer**". Nichtbeachtung verletzt bestehendes Urheberrecht.

INHALT

I. DIE ENTWICKLUNGS-UMGEBUNG VON KICK-PASCAL

II. SYNTAX

III. DIE DATENTYPEN VON KICK-PASCAL

IV. VARIABLEN UND KONSTANTEN

V. PROZEDUREN UND FUNKTIONEN

EINFÜHRUNG IN DIE AMIGA-SYSTEMPROGRAMMIERUNG

KICK-PASCAL V2.0
Die neue Version im Überblick

Im Dezember 1989 erschien die erste Version von KICK-PASCAL und machte das Programmieren in PASCAL auf dem AMIGA populär. Nun ist ein erstes größeres Upgrade des KICK-PASCAL Compiler-Systems erhältlich.

Es wurden viele nötige bzw. wünschenswerte Ergänzungen vorgenommen. Neben vielen Details sind folgende Features hinzugekommen:

- Programme können in handliche UNITS und MODULE zerlegt werden. Ein in das PASCAL-System integrierter schneller LINKER fügt solche getrennt compilierbare Programmteile Module zusammen. Da der Linker im großen und ganzen zum AMIGA-Standardlinker "ALink" kompatibel ist, können auf diesem Wege auch Assembler-Routinen eingebunden werden.
- Der Compiler beherrscht jetzt auch 64 Bit breite Fließkommazahlen. Dadurch sind REAL-Berechnungen mit einer Genauigkeit von 15 bis 16 Dezimalstellen bei einem Exponentenbereich von 1E±308 möglich (bisher gab es nur REALs einfacher 7-stelliger Genauigkeit mit Werten zwischen ca. 1E+17 und 1E-18).
- Der von KICK-PASCAL erzeugte Programmcode ist besser geworden. Er war zwar bisher auch schon ganz akzeptabel, aber jetzt ist er ehrlich gut.
- Ein Exception-Handler fängt einfache Abstürze ab. Wenn man einmal vergessen hat, eine Library zu öffnen oder einen Zeiger sinnvoll zu initialisieren, kann man nun in der Regel nach einer Laufzeit-Fehlermeldung weiterarbeiten, als sei nichts geschehen.
- Der Editor wurde überarbeitet. Beispielsweise ist es jetzt möglich, die Tastenbelegung weitgehend frei zu wählen.
- In einer Konfigurationsdatei werden zahlreiche Einstellungen des KICK-PASCAL-Systems festgehalten.

REFERENZ

KAPITEL I

DIE ENTWICKLUNGS-UMGEBUNG VON KICK-PASCAL

I. Die Entwicklungs-Umgebung von KICK-PASCAL

1.1 Installation

KICK-PASCAL wird auf zwei Disketten ohne Kopierschutz geliefert, so daß das System ohne Probleme auf einer Festplatte oder einer Workbench-Diskette installiert werden kann. Als erstes sollten Sie sich deshalb Sicherheitskopien der Original-Disketten anlegen.

Die wichtigste Datei auf der Disk#1 ist KICK-PASCAL. Dieses Programm enthält das komplette Editor-Compiler-Linker-System. Gemessen an seinem Leistungsumfang, ist das Programm sehr kompakt, so daß es auch auf einer Workbench-Diskette leicht unterzubringen ist.

Das Verzeichnis "Include" enthält die für die Systemprogrammierung unverzichtbaren Includefiles. Dieses Verzeichnis sollten Sie an einer sinnvollen Stelle auf Ihrer Festplatte oder PASCAL-Arbeitsdiskette, falls dort genug Platz ist, unterbringen. Im Verzeichnis "Unit" stehen fertig übersetzte Units, die man mittels "Uses"-Anweisung in eigene Programme einbinden kann. Sie sind für das Arbeiten mit KICK-PASCAL nahezu unverzichtbar und sollten auf jeden Fall installiert werden.

Tips für Disketten-Benutzer: Am bequemsten ist es wohl, sich eine bootfähige KICK-PASCAL-Arbeitsdiskette zusammenzustellen. Kopieren Sie die Workbench-Disk, installieren Sie darauf das KICK-PASCAL- Programm und setzen Sie gleich den Aufruf des PASCAL-Systems in die Startup-Sequence: "Run KICK-PASCAL". Da KICK-PASCAL keinerlei Ein- oder Ausgaben über die Standardausgabe durchführt, können Sie es auch mit "RunBack KICK-PASCAL" oder (unter Arp 1.3) mit "ARun KICK-PASCAL NOIO" starten - auf diese Weise können Sie das CLI-Fenster nachher auch schließen.

Um die Units unterzubringen, müssen Sie wohl ein wenig auf Ihrer Disk aufräumen (überflüssige Fonts, Keymaps und Druckertreiber löschen usw.). Die "mathieeedoubbas.library" müssen Sie aber unbedingt behalten, denn der Compiler braucht sie. Auch die anderen "mathxxxx"-Librarys brauchen Sie noch, wenn Sie in Ihren PASCAL- Programmen mit Real-Zahlen arbeiten wollen.

Die Include-Files sind nicht so wichtig, da sie zum größten Teil in Form von vorcompilierten Units vorliegen. Sie benötigen sie aber um z.B. Änderungen an den Units vorzunehmen oder um für KICK-PASCAL 1.0 geschriebene Programme überrsetzen zu können. Deshalb sollten Sie, sofern Sie zwei Diskettenlaufwerke haben, eine Diskette formatieren, die Sie beim Arbeiten mit KICK-PASCAL immer in das zweite Laufwerk einlegen und auf der Sie die Includefiles sowie später Ihre eigenen PASCAL-Quelltexte ablegen.

Wenn Sie nur ein Diskettenlaufwerk besitzen, wird es allerdings etwas eng. In diesem Fall ist es eine elegante Lösung (d. h. ohne übermäßiges Diskettenwechseln),

auf einer Boot-Diskette das Programm KICK-PASCAL zu installieren (und evtl. in der Startup-sequence zu starten) und auf einer zweiten (nicht notwendigerweise bootfähigen) Diskette den Include- und den Unit-Ordner sowie später Ihre eigenen Quelltexte unterzubringen.

Das Programm "PASCALPrefs" sollten Sie sich ebenfalls auf einer Ihrer Disketten installieren, auch wenn Sie es nicht allzuoft und später, wenn alles optimal eingestellt ist, überhaupt nicht mehr benötigen werden. Die zahlreichen Quelltexte und lauffähigen Programme auf der zweiten Disk müssen Sie als Diskettenuser nicht unbedingt auf Ihren Arbeitsdisketten installieren, es genügt, wenn Sie eine Sicherheitskopie davon anlegen.

Für Festplatten-Besitzer: Am besten legen Sie ein Verzeichnis an, z. B. unter dem Namen KICK-PASCAL, und ziehen dann zunächst einmal beide Disketten vollständig 'rüber:

```
copy PASCAL:#? DH0:KICK-PASCAL all
copy PASCALSource:#? DH0:KICK-PASCAL all
```

Später können Sie ja immer noch löschen, was Sie nicht mehr brauchen.
Viele Demoprogramme greifen auf PASCAL: zu. Um sie von der Harddisk starten zu können, sollten Sie mit assign PASCAL: DH0:KICK-PASCAL l ein entsprechendes logisches Device einrichten (vorher die gleichnamige Diskette aus dem Laufwerk nehmen). Diesen Befehl können Sie bei Bedarf auch in Ihre Startup-Sequence setzen.

Ferner finden Sie auf Ihrer Disk die "rct.library". Diese wird für das RCT-Demoprogramm benötigt und sollte deshalb in das Libs:- Verzeichnis kopiert werden.

Nun ist das System im Prinzip startklar - einmal abgesehen von einer Meldung, die beim Programmstart erscheint:

```
"Can't open configuration file."
```

KICK-PASCAL erwartet eine Konfigurationsdatei namens "PASCAL.config" im "S:"-Verzeichnis. Es läuft zwar auch, wenn diese Datei nicht vorhanden ist, benutzt dann aber Default-Einstellungen.

Deshalb sollten Sie erst einmal das Programm "PASCALPrefs" von der Diskette 1 starten und damit eine Datei namens "S:PASCAL.config" erzeugen lassen. Das Programm ist dabei selbsterklärend, und vorerst müssen Sie noch keine besonderen Einstellungen vornehmen. Alles, was zu tun bleibt, ist das Anlegen dieser Datei mit zwei Mausklicks zu bestätigen und "PASCALPrefs" dann mit einem weiteren Klick zu verlassen.

Besser ist es aber, wenn Sie vorm Abspeichern noch den Menüpunkt "Defaultwerte definieren" anklicken. Sie gelangen dann in eine Art Menü, wo Sie verschiedene Einstellungen vornehmen können. Unten finden Sie dort vier Text-Gadgets, in denen Sie die Directories eintragen können, in denen die Include-Dateien bzw. die Units stehen.

Wenn Sie knapp an Speicher sind (z. B. 512 kByte), sollten Sie nur die beiden linken Gadgets benutzen und dort die Pfade eintragen, während Sie den Inhalt der beiden rechten Gadgets löschen. Wenn aber Speicher bei Ihnen kein Problem darstellt (z. B. 1 MByte), lassen Sie die linken Gadgets unverändert (dort müßte etwas von "RAM:xxx" stehen) und tragen die Pfade rechts ein. Das bedeutet dann, daß einmal gelesene Dateien zwecks nochmaliger Benutzung in das erste Verezichnis (in der schnellen RAM-Disk) umkopiert werden, was die Arbeitsgeschwindigkeit enorm steigert - doch dazu später mehr.

Als Diskettenbenutzer sollten Sie hier auf keinen Fall die Namen von Laufwerken ("DF0:", "DF1:", ...), sondern die Namen der Disketten in den Pfaden benutzen, um einen Zugriff auf die richtige Diskette zu gewährleisten.

Nach diesen ersten Einstellungen speichern Sie die so geänderte Konfiguration wie oben beschrieben ab und verlassen das Programm "PASCALPrefs". Eine vollständige Beschreibung der diversen anderen Einstellmöglichkeiten folgt später.

1.2 Programmstart und erste Schritte

KICK-PASCAL kann wahlweise von der Workbench oder vom CLI gestartet werden. Nach dem Start erscheint zuerst immer das Titelbild mit der Frage, wieviel Arbeitsspeicher reserviert werden soll. Das Programm sorgt stets dafür, daß 50 KByte Chip-Memory frei bleiben. Trotzdem sollten Sie den angegebenen Maximalwert nur dann wählen, wenn es unbedingt sein muß. In der Regel reicht es, wenn Sie den vorgegebenen Defaultwert von 80 KByte einfach mit Return bestätigen.

Wenn Sie das PASCAL-System vom CLI bzw. von einer Shell aus starten, können Sie optional einige Parameter angeben: Als erstes wäre da die Angabe eines Dateinamens möglich, z. B.

```
KICK-PASCAL PASCALSource:Turtlegrafik/Törteltest2.p
```

Die betreffende Datei wird dann automatisch in den Editor geladen. Wenn der Dateiname wie hier mit ".p" endet, wird das Programm außerdem automatisch compiliert.

Dann gibt es noch drei Einstellmöglichkeiten: "w" (Workspace) für die Wahl der Arbeitsspeichergröße, "c" (Config) für die Benutzung einer anderen Konfigurationsdatei als "S:PASCAL.config" und "m" (Main file) zum Definieren einer sog. Hauptdatei. Die Reihenfolge dieser Parameter ist beliebig, in die Syntax ist immer wahlweise

`x=Parameter` oder `-xParameter` oder `-x=Parameter`

Dabei ist stets zu beachten, daß vor dem Parameter kein Leerzeichen stehen darf. Beispiele:

```
KICK-PASCAL -w50 PASCALSource:Datei.p
```

Bei 50 kByte Workspace wird die angegebene Datei geladen und, wegen der Endung des Namens, automatisch compiliert.

```
KICK-PASCAL c=PASCAL:Alternativ.config m=PAS:Haupt.p
```

KICK-PASCAL wird normal gestartet, aber mit einer anderen Konfigurationsdatei und einem vorgewählten Main-File.

Der Parameter "-w" bzw. "w=" ohne folgende Speichergröße bewirkt, daß der in der Konfigurationsdatei eingestellte Defaultwert benutzt wird.
Wenn Sie "-c" bzw. "c=" ohne Dateinamen verwenden, wird keine Konfigurationsdatei geladen, d. h. KICK-PASCAL verwendet die normalen Defaultwerte.

Nach dem Start und der Wahl der Arbeitsspeichergröße landen Sie automatisch im komfortablen Editor. Hier können Sie Ihre PASCAL-Programme schreiben und bearbeiten. Eine ausführliche Erläuterung der Editor-Funktionen folgt später, deshalb will ich mich hier auf den Hinweis beschränken, daß RETURN, Delete, Backspace und die Cursortasten so funktionieren, wie Sie es gewohnt sind.

Am besten probieren Sie zuerst etwas herum. Sie könnten z. B. das folgende erste PASCAL-Programm abtippen. Nein, nicht alle Handbuch-Autoren sind phantasielos. Das Hallo-Programm ist ganz einfach Tradition:

```
PROGRAM Hello;
BEGIN
  writeln('Hello World!')
END.
```

Nun haben Sie ein Programm als Quelltext vorliegen und können sich mit den Compiler-Funktionen vertraut machen. Drücken Sie dazu am besten die Taste F9. Dann wird - vorausgesetzt Ihr Programm ist syntaktisch korrekt - folgendes ausgegeben:

```
Compiling...
Linking...
Ready.
Running.
Hello World!
>
```

Wenn der Compiler zuerst einmal kurz auf die Diskette oder Festplatte zugegriffen hat, so lag das daran, daß er eine Bibliothek (die "mathieeedoubbas.library") laden mußte. Das tut er aber nur beim ersten Mal - bei allen nachfolgenden Compilevorgängen können Sie die volle Geschwindigkeit von KICK-PASCAL genießen.

Sie befinden sich jetzt (erkennbar am Prompt ">") im Tastaturmenü, das im folgenden Kapitel genau beschrieben wird. Mit der ESC-Taste gelangen Sie von hier wieder zurück in den Editor.

Fast alle Funktionen von KICK-PASCAL können wahlweise mit Maus oder Tastatur aufgerufen werden. Da die wichtigsten Pull-Down-Menüs wohl keine besondere Erläuterung erfordern, wollen wir uns zuerst mit der Bedienung über die Tastatur befassen.

Neben den Pull-Down-Menüs gibt es noch ein Tastaturmenü. Vom Editor können Sie auf vier verschiedene Arten dorthin gelangen. Normalerweise werden Sie wohl einfach die ESC-Taste drücken. Falls Sie aber die Tasten noch nicht auswendig kennen, können Sie auch mit der Maus das Pull-Down-Menü "Info/Hilfe" anwählen, in diesem Fall werden dann ebenfalls die Menüpunkte ausgegeben.
Die beiden anderen Möglichkeiten: Drücken Sie vom Editor aus - wie oben bereits beschrieben - die Taste F9 wahlweise mit oder ohne "Shift". Im ersten Fall wird Ihr Programm compiliert, im zweiten Fall gestartet und zuvor automatisch übersetzt, falls das nicht bereits geschehen sein sollte. Anschließend landen Sie jedenfalls im Tastaturmenü - falls der Compiler nicht einen Fehler gefunden hat, denn in diesem Fall gelangen Sie wieder in den Editor zurück.

1.3 Das Tastaturmenü

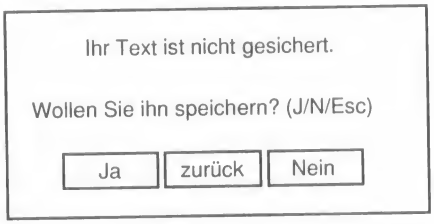
Nun sind Sie also auf eine der oben beschriebenen Arten in das Tastaturmenü gelangt. Das erkennen Sie an dem Prompt ">" vor dem Cursor. Zunächst sollten Sie sich vielleicht das Menü ausgeben lassen, sofern Sie nicht sowieso schon mit dem "Hilfe"-Menü hierher gesprungen sind, denn dann steht es ja schon da. Auch dazu gibt es wieder einmal mehrere Möglichkeiten; generell wird Ihnen noch auffallen, daß viele Funktionen des KICK-PASCAL-Systems auf mehrere Arten aufgerufen werden können - für jeden und jede Situation etwas. Zum einen gibt es natürlich das bereits erwähnte Pull-Down-Menü "Info/Hilfe". Aber der Sinn des Tastaturmenüs liegt ja darin, daß man hier ohne Maus auskommt. Also gibt es auch zwei verschiedene Tasten dafür: wahlweise "Help" oder "CR".
Unter der Zeile mit dem Programmnamen stehen jetzt die 15 Menüpunkte:

Load	**S**ave	save **A**s	**O**bjectfile	**E**xefile
ESC=edit	**C**ompile	**R**un	**F**ind error	**P**references
?=info	this Menu	prin**T**	**N**ew	**Q**uit

Dabei wird jeweils das Zeichen, das Sie tippen müssen, rot ausgegeben. Die Zeilen darunter geben Ihnen Aufschluß über die Speicherbelegung, dazu später mehr. Darunter finden Sie wieder den Cursor samt Prompt ">". Wenden wir uns nun also den einzelnen angebotenen Punkten zu.

1.3.1 Load

Damit können Sie eine Datei laden. Falls Sie im Editor bereits einen Text geschrieben oder bearbeitet haben, ohne ihn abzuspeichern, öffnet sich zuerst ein kleines Fenster mit der Meldung:



Übrigens erscheint dieser Requester jedesmal, wenn Sie kurz davor stehen, einen seit der letzten Veränderung nicht mehr oder noch gar nicht abgespeicherten Text zu verlieren. Ich werde dann aber nicht mehr ausdrücklich darauf hinweisen.

In dieser Situation haben Sie wieder einmal die Wahl: Entweder klicken Sie mit der Maus eines der Gadgets an, oder Sie tippen eine der Tasten "J", "N" oder "Esc". Bei "Ja" erscheint einer der jedem AMIGA-Anwender wohlbekannten Datei-Requester, mit dem Sie komfortabel angeben können, unter welchen Namen Ihr Text abgespeichert werden soll. "Nein" bedeutet stets, daß der Text im Editor verloren gehen soll. Mit "Zurück" können Sie jeweils Ihre Wahl rückgängig machen: in diesem Fall bedeutet das, daß Sie doch nichts laden wollen.

Zurück zum Laden von Texten: Am unteren Window-Rand erscheint nun die Frage "Filename:". Einleuchtenderweise können Sie jetzt einen Dateinamen eintippen (dabei stehen Ihnen übrigens im wesentlichen dieselben Bearbeitungsmöglichkeiten wie im Editor zur Verfügung) und mit RETURN bestätigen, oder die "Esc"-Taste drücken, falls Sie es sich anders überlegt haben.

1.3.2 Save

Der Text wird abgespeichert. Falls er noch keinen Namen hat (dann steht in der Titelzeile des Editors am Anfang einfach "EDITOR", und im Menü steht hinter "Filename" überhaupt nichts), wird zunächst einmal genau wie bei "Load" nach dem gewünschten Dateinamen gefragt.

Dieser Name wird dann auch bei jedem folgenden "Saven" benutzt.

1.3.3 Save As

Dieser Menüpunkt unterscheidet sich von "Save" nur dadurch, daß immer zuerst nach dem Dateinamen gefragt wird, auch wenn Ihr Text schon einen Namen hat.

1.3.4 Objectfile

Hier hat sich eine Änderung gegenüber KICK-PASCAL Version 1.0 ergeben: Der Menüpunkt, der in jener Version diese Bezeichnung trägt, heißt jetzt "Exefile". Die Funktion "Objectfile" schreibt - vorausgesetzt, es liegt ein Programm in compilierter Form vor - den Programmcode in eine Datei, allerdings nicht in einer sofort lauffähigen Form, sondern in dem Format, das der Linker benötigt, um Module zu einem Programm zusammenzubinden. Im Kapitel über Module wird noch näher auf diesen Menüpunkt eingegangen werden.

Das Abspeichern der Object-Datei zerstört den Code, Sie müssen also bei Bedarf neu compilieren lassen.

1.3.5 Exefile

Wenn Sie ein Programm in compilierter (also lauffähiger) Form vorliegen haben, können Sie mit dieser Funktion eine vom KICK-PASCAL-System unabhängige, sofort lauffähige Programmdatei erzeugen lassen. Der Dateiname wird in gewohnter Manier erfragt.

Falls Ihr Quelltext bereits einen Namen hat, wird hier ein geeigneter Name vorgeschlagen, den Sie bei Gefallen nur noch mit RETURN bestätigen müssen. Wenn der Name des Texts eine mit einem Punkt abgetrennte sog. "Extension" wie z. B. ".P" oder ".PAS" hat, wird diese abgeschnitten (aus "Hallo.p" wird also schlicht "Hallo"), andernfalls wird ".exe" an den Namen angehängt.

1.3.6 Compile

Dies ist gleichbedeutend mit der Tastenkombination Shift-F9, die man natürlich auch vom Tastaturmenü aus benutzen kann. Das Programm wird compiliert und gelinkt. Der Übersetzungsvorgang kann mit der Taste F10 abgebrochen werden. In diesem Fall oder wenn der Compiler einen Fehler festgestellt hat, gelangen Sie zurück in den Editor. Die Fehlermeldung steht dann in den untersten Bildschirmzeilen und ein Pseudo-Cursor blinkt an der fehlerhaften Stelle. Nun drücken Sie die Taste "Esc" und können den Fehler berichtigen.

Wenn das Programm fehlerfrei übersetzt und gelinkt werden konnte, wird die Speicherbelegung ausgegeben.

1.3.7 External

Mit der Taste "x" können Sie eine Datei übersetzen, ohne sie in den Editor zu laden oder den Text im Editor zu verlieren. Sie werden hier nach dem Dateinamen gefragt (mit "Esc" kommen Sie wie immer wieder 'raus). Nach erfolgreichem Übersetzen können Sie das Programm mit "Run" (siehe unten) starten, Exe-Datei erzeugen usw.

1.3.8 Run

Dieser Menüpunkt entspricht der Taste F9: Das Programm wird gestartet. Falls es noch nicht oder seit der letzten Änderung im Quelltext nicht mehr übersetzt wurde, wird zuvor automatisch der Compiler aufgerufen.

1.3.9 Find Error

Wenn Ihr mit "Run" gestartetes Programm mit einer Laufzeit-Fehlermeldung abgebrochen ist oder Sie es mit der F10-Taste abgebrochen haben (denn auch das ist immer möglich, sofern Sie die entsprechende Compiler-Option nicht ausgeschaltet haben - später dazu mehr), stellt das KICK-PASCAL-System die Codeadresse fest, an der die Programmausführung abbrach und gibt sie zusammen mit der obligatorischen Speicherbelegung aus. Mit der "F"-Funktion können Sie sich dann die dazu gehörige Stelle im Quelltext suchen lassen. Dazu einige Anmerkungen:

- Die gefundene Adresse ist nicht immer richtig. Vor allem ist sie dann unsinnig, wenn der Programmabbruch in einem eingelinkten Modul erfolgte.
- In der Regel steht der Cursor hinter dem Befehl, bei dem der Fehler auftrat, unter Umständen am Anfang des nächsten Befehls.
- Der Fehler wird gesucht, indem das Programm ganz normal compiliert wird, wobei der Compiler lediglich ständig prüft, ob er die gesuchte Stelle schon erreicht hat. Deshalb geht der lauffähige Code durch das Suchen verloren, so daß bei Bedarf neu compiliert werden muß.

1.3.10 Main File

Sie können hier einen Dateinamen eingeben, der dann über dem Menü (unter "Filename") angezeigt wird. Nach dem Compilieren eines Units wird diese Hauptdatei automatisch extern compiliert - mehr dazu im Kapitel "Units und Module".

1.3.11 Preferences

Mit diesem Menüpunkt gelangen Sie in ein kleines Untermenü, mit dem Sie verschiedene Compiler-Optionen einstellen können:

- **Test subrange:** Die Grenzen von Ausschnittstypen sind zur Laufzeit bei Wertzuweisungen zu prüfen.
- **Index range:** Bei Array-Zugriffen sind die Grenzen zu testen.
- **Breakpoints:** An geeigneten Stellen wird im Code eine Überprüfung der F10-Taste eingebaut, so daß das Programm mit dieser Taste abgebrochen werden kann.
- **Stacksize:** Ein Überlauf des Stacks ist abzufangen.
- **Arithmetic overflow:** Bei Rechenoperationen jeder Art sind Überläufe abzufangen.

Als Default sind alle Optionen bis auf die letzte angewählt. Jede dieser Option kann mit ihrem Anfangsbuchstaben an- und ausgeschaltet werden. "Q" schaltet alle Optionen aus, dadurch wird das Programm schneller, deshalb die Bezeichnung "Quick".

Mit "Esc" gelangen Sie in das normale Tastaturmenü zurück. Die eingestellten Optionen werden aber erst wirksam, wenn das Programm wieder compiliert wird. Deshalb wird der Programmcode - falls vorhanden - bei jeder Änderung der Preferences gelöscht.

Alternativ kann man die Preferences auch über das Pull-Down-Menü "Optionen/ Compiler" einstellen. Ferner können sie im Quelltext durch Compileranweisungen der Form {$opt ... } eingestellt werden. Eine genauere Erläuterung der einzelnen Optionen finden Sie deshalb im Kapitel über "Compiler Directives".

1.3.12 Print

Das Druckermenü ermöglicht es Ihnen, Ihre Quelltexte bequem ein- oder zweispaltig mit Seitennumerierung zu drucken. Es erscheint ein kleiner Requester, den Sie mit der Esc-Taste oder dem Fensterschließsymbol verlassen können. Im obersten Gadget wird nach dem Dateinamen gefragt.

Als Default ist "Prt:", also der Drucker, vorgegeben. Sie können den Namen aber auch ändern, z. B. um das Listing zunächst einmal in eine Datei umzuleiten. Darunter können Sie eintragen, wieviele Zeichen der Drucker pro Zeile im Normal- bzw. Schmaldruck schreiben kann und soll. Ganz unten sind vier Gadgets, die mit der Maus oder mit der angegebenen Zifferntaste angewählt werden und den Druckmodus festlegen:

1. **Einspaltiger Druck**
   Der Text wird ganz normal zum Drucker geschickt, abgesehen davon, daß auf jeder Seite als Kopf der Dateiname und die Seitennummer gedruckt werden.
2. **Schmaldruck**
   Der Drucker wird hierbei in den Modus Elite-Condensed geschaltet. Es ist aber vom Drucker abhängig, wieviele Zeichen er in einer Zeile unterbringt. Deshalb müssen Sie evtl. vorher den Eintrag im Gadget "Spalten bei Schmaldruck" ändern (der Wert für "Spalten bei Normaldruck" wird den Preferences entnommen und müßte deshalb stimmen).
3. **Zweispaltiger Druck bei normaler Schriftbreite**
   Dabei stehen für jede Spalte normalerweise weniger als 40 Zeichen pro Zeile zur Verfügung, so daß dieser Modus solchen Texten vorbehalten bleiben sollte, die wirklich im wesentlichen nicht mehr Zeichen pro Zeile haben.
4. **Zweispaltiger Druck bei doppelter Dichte**
   Dies ist wohl der beste (platzsparendste) Druckmodus für Listings.

Mit der "Esc"-Taste können Sie jederzeit den Druckvorgang abbrechen.

1.3.13 New

Dieser Menüpunkt löscht den Text im Editor vollständig.

1.3.14 Quit

Dies ist eine der Möglichkeiten, das KICK-PASCAL-System zu verlassen. Andere Möglichkeiten sind das gleichnamige Pull-Down-Menü, das Close-Gadget des Fensters oder ein Tastendruck im Editor, falls Sie die Tastenbelegung entsprechend eingestellt haben.

1.3.15 Esc: Edit

Mit der Esc-Taste gelangen Sie zurück in den Editor.

1.3.16 CR: This Menu

Mit "CR"- oder der "Help"-Taste können Sie sich, wie oben bereits erwähnt, das Menü ausgeben lassen.

1.3.17 Info

Mit dem Fragezeichen erhalten Sie Inforamtionen über Autor und Copyright.

1.3.18 Space-Taste

Mit der Leertaste können Sie sich jederzeit die Belegung des Arbeitsspeichers und Informationen über ein compiliertes Programm ausgeben lassen.

1.4 Die Pull-Down-Menüs

Nach diesem Kapitel für Tastaturfreaks wollen wir uns nunmehr dem eher eines AMIGA würdigen Teil der Benutzeroberfläche von KICK-PASCAL zuwenden. Wahrscheinlich haben Sie, wenn Sie dieses hier lesen, die meisten Pull-Down-Menüs schon längst ausprobiert. Trotzdem werde ich noch einmal ausführlich auf alle Menüs eingehen:

1.4.1 Projekt

Es ist eine auf dem AMIGA weit verbreitete Konvention, ganz links in die Menüleiste ein Menü dieses Namens und Inhalts zu setzen. Schon allein deshalb gibt es zu diesem Menü nicht viel zu sagen. Die einzelnen Punkte unterscheiden sich von ihren Äquivalenten im Tastaturmenü nur dadurch, daß hier Dateinamen mit einer der bekannten und geschätzten Datei-Auswahlboxen erfragt werden.

Der Requester ist so eingestellt, daß alle Dateien, außer denen mit der Endung ".info" (also den Icons), angezeigt werden. Mit dem Gagdet unten in der Mitte kann der Requester so umgeschaltet werden, daß ausschließlich Dateien mit der Endung ".p" angezeigt werden, der empfohlenen Endung für KICK-PASCAL-Quelltexte.
Als kleine Besonderheit kann man ein anderes Gerät oder Verzeichnis anwählen, bevor das alte vollständig eingelesen wurde. Außerdem kann der Filerequester vollständig mit der Tastatur bedient werden: Mit der Returntaste können Sie die Operation bestätigen (sofern ein Dateiname gewählt wurde), "Esc" entspricht dem Cancel-Gadget und "Tab" dem Gadget "*.p". Mit den Tasten "P" und "D" aktiviert

man die Textgadgets "Pfad" bzw. "Datei", während die Funktionstasten den zehn Geräte-Gadgets entsprechen.

Sobald das Verzeichnis vollständig eingelesen worden ist, wird der oberste Eintrag farblich vervorgehoben. Diesen "Cursor" können Sie dann mit den Cursortasten durch das Directory bewegen und mit der Leertaste den gerade hervorgehobenen Eintrag anwählen.

1.4.2 Editor

Dieses Menü enthält im wesentlichen Funktionen, für die es auch Editor-Tastenkombinationen gibt. "Gehe nach" setzt den Cursor an den Anfang bzw. das Ende des Quelltextes. Das Untermenü "Block" stellt die elementaren Blockoperationen zur Verfügung - siehe auch Kapitel 1.5.2: "Block-Operationen".

Mit den Punkten des Untermenüs "Datei" kann man einen Textblock in eine Datei schreiben oder aus einer Datei lesen und dann wahlweise ("Block lesen") an der Cursorposition oder ("Anhängen") am Ende des Textes einfügen. Für den letzten Punkt gibt es übrigens kein Tastatur-Äquivalent. Der Dateiname wird hier jeweils mit einer Dateiauswahlbox abgefragt, während er bei den entsprechenden Tastenkombinationen (siehe nächstes Kapitel) eingetippt werden muß.

"Block schieben" ist einfach eine Menü-Alternative für die Tasten F5 und F6, mit denen man einen zuvor zu markierenden Block seitlich verschieben kann. Auf die Menüpunkte "Finden" und "Ersetzen" wird in einem eigenen Kapitel eingegangen werden. "Wiederhole" führt die zuletzt durchgeführte Finden-/Ersetzen-Operation noch einmal aus.

1.4.3 Starten

Die beiden ersten Funktionen dieses Menüs, "Übersetzen" und "Starten", sind identisch mit der Taste "F9" mit bzw. ohne "Shift" und den Operationen "Compile" bzw. "Run" des Tastaturmenüs. Auch der letzte Punkt, "Fehlersuche", ist nur ein Analogon zur Taste "F" (also "Find Error") des Tastaturmenüs, d. h. nach einem Laufzeit-Fehler eines PASCAL-Programms kann hiermit die Abbruchstelle im Quelltext gesucht werden. "Hauptdatei" und "Compile Datei" unterscheiden sich von den Tastaturkommandos "Main file" bzw. "External" nur dadurch, daß hier der Dateiname mit dem Filerequester erfragt wird.

Neu ist dagegen der Schalter "Parameter", den man nicht mit der Tastatur bedienen kann. Wenn ein Programm vom CLI aus startet, kann man ihm bekanntlich eine Parameter-Zeile übergeben. So ist z. B. bei "Dir DF1:x opt a" die Zeichenfolge "DF1:x opt a" Parameter des Programms "Dir". Auch KICK-PASCAL-Programme können solche Parameter übernehmen. Nun will man als Programmierer natürlich schon während der Programmentwicklung, also beim Programmstart aus der Entwicklungsumgebung, testen, ob das Programm das Gewünschte tut.

Ist nun die Option "Parameter" auf "Ein" geschaltet, wird in Zukunft vor jeden Programmstart gefragt, was dem PASCAL-Programm als "simulierter" Parameter übergeben werden soll.

1.4.4 Optionen

Dieses Menü enthält verschiedenste globle Einstellmöglichkeiten für alle Teile des KICK-PASCAL-Systems:

1.4.4.1 Compiler

Dieses Untermenü entspricht dem "Preferences"-Tastaturmenü.

1.4.4.2 Linker

Der integrierte Linker von KICK-PASCAL ist im wesentlichen zum AMIGA-Standard-Linker "ALink" (bzw. der besseren PD-Version "BLink") kompatibel. Jener Linker erkennt Bezeichner nur dann als identisch, wenn sie in ihrer Groß-/Kleinschreibung übereinstimmen. Beispielsweise sind "Test" und "test" zwei verschiedene Namen. Als PASCAL-Programmierer ist man aber gewohnt, daß die Schreibweise von Bezeichnern keine Rolle spielt. Deshalb haben Sie durch dieses Menü die Wahl, ob der integrierte KICK-PASCAL-Linker sich PASCAL-mäßig verhalten ("GROSS/klein egal") oder sich an den "ALink"-Standard halten soll ("GROSS/klein beachten").

1.4.4.3 Spezial

Dieses Untermenü enthält in der vorliegenden Version nur eine Funktion: Das automatische Ergänzen von fehlenden Semikola.

Wie Sie vielleicht schon wissen, kommt KICK-PASCAL in der Regel ohne Semikola ";" als Trennzeichen aus. Wenn Sie dies aber ausnutzen, ist Ihr Programm nicht mehr ohne weiteres auf andere Rechner und Compiler übertragbar. Deshalb können Sie sich fehlende Semikola automatisch ergänzen lassen. Der Compiler merkt sich dann jeweils, wo ein solches Zeichen fehlt, und ergänzt es, sobald er das Programm fehlerfrei übersetzt hat. Er hat dafür aber nur einen begrenzten Pufferspeicher zur Verfügung, so daß bei jedem Compilerdurchlauf nur maximal 25 ";" " eingefügt werden. Außerdem wird diese Operation nur in dem Text im Arbeitsspeicher durchgeführt, also nicht in Include-Files.

Da es wohl nicht jedermanns Sache ist, wenn der Compiler im Source Änderungen vornimmt, muß man diese Funktion bei Bedarf mit dem Pull-Down-Menü "Optionen/Spezial/Ergänze Semikola" einschalten.

1.4.4.4 Einfügen

Entspricht der Tastenkombination Ctrl-V: Der Editor wird zwischen Einfüge- und Überschreibmodus umgeschaltet.

1.4.4.5 Protokoll

Wird ein Programm vom PASCAL-System aus bei eingeschaltetem Protokollmodus gestartet, so werden alle Ausgaben des Programms und Eingaben des Benutzers, die über die Standard-Ein-/Ausgabe erfolgen, in eine Datei oder zum Drucker kopiert (also nicht bloß umgeleitet - das Programm läuft ganz normal).

1.4.4.6 Sprache

Das KICK-PASCAL-System ist größtenteils zweisprachig (Englisch und Deutsch). Mit diesem Menu können Sie die Sprache wählen.

Übrigens: KICK-PASCAL erkennt automatisch, welche Sprache vom Benutzer gewünscht wird, indem es nachsieht, auf welcher Taste das "Z" liegt.

1.4.4.7 Dateien

Dieses Untermenü enthält verschiedene Einstellmöglichkeiten für das Abspeichern von Quelltexten: Wenn die Option "Autosave" aktiviert ist, wird der Quelltext vor jedem Programmstart abgespeichert, sofern er seit der letzten Änderung noch nicht gesichert wurde.

"Backup" bewirkt, daß vor dem Saven des Quelltexts eine evtl. vorhandene gleichnamige Datei (also die zuletzt abgespeicherte Version, die jetzt überschrieben werden würde) durch Anhängen von ".backup" umbenannt wird. Dies erhöht die Datensicherheit enorm, und auch die Lebensdauer von Disketten wird erhöht, da der Text jetzt nicht mehr bei jedem Abspeichern auf dieselben Disk-Blöcke geschrieben wird. Deshalb sollte diese Option nur im äußersten Notfall (Disk voll oder so) abgeschaltet werden.

Mit "Info" können Sie festlegen, ob KICK-PASCAL beim Schreiben von Dateien aller Art ein entsprechendes Icon anlegen soll.

Defaulteinstellung ist hier, daß dieser Punkt nur beim Start von der Workbench angewählt ist. Aber Sie können das auch in der Konfigurationsdatei einstellen.
In den Icons zu Textdateien wird das KICK-PASCAL-System als "Default Tool" und die Workspace-Größe als "Tool Type" eingetragen. Dadurch gelangen Sie beim Anwählen eines solchen Icons auf der Workbench automatisch ins KICK-PASCAL.

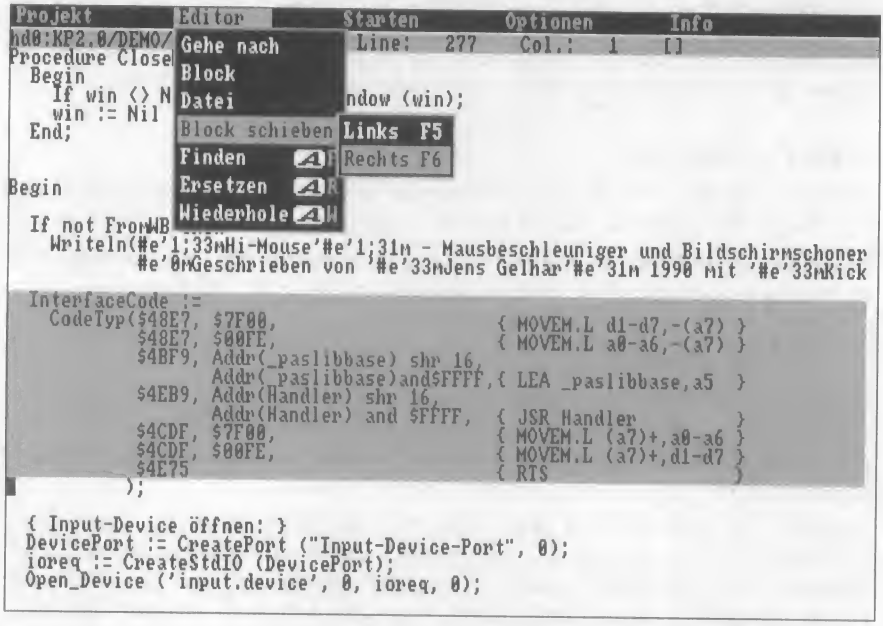
1.4.4.8 Suchpfade

Es erscheint ein kleiner Requester, in dem Sie eintragen können, von wo das KICK-PASCAL-System Includedateien und Units lesen soll. Mehr dazu in den Kapiteln über Includes bzw. Units.

1.4.5 Info

Dieses Menü gibt die üblichen Infos aus, bei "Version" können Sie die Versionsnummer nachlesen, und mit "Hilfe" gelangen Sie in das Tastaturmenü.

1.5 Der Editor



Mit dem Editor schreibt und bearbeitet man PASCAL-Quelltexte. Er kann auch für alle anderen Arten von ASCII-Files benutzt werden.

Die Tastaturbelegung des Editors kann mit dem Programm "PASCALPrefs" weitgehend frei gewählt werden. In diesem Abschnitt wird auf die Standardbelegung der Editortasten eingegangen. Wem sie nicht zusagt, der darf sie natürlich ändern.

Die oberste Zeile des Windows ist farblich hervorgehoben und enthält verschiedene Informationen: Links steht der Dateiname des Textes oder einfach "EDITOR", falls der Text noch keinen Namen hat. Daneben steht jeweils entweder "Insert" oder "Overwrite", je nach dem, ob der Editor sich gerade im Einfüge- oder Überschreibmodus befindet. Hinter "Line:" und "Col.:" können Sie die Zeilen- und Spaltennummer des Cursors ablesen. Rechts davon werden ein oder zwei eckige Klammern angezeigt, wenn ein Block markiert ist.

Auch im Editor können die wichtigsten Funktionen auf mehrere verschiedene Arten aufgerufen werden, so daß für jeden Geschmack das Richtige dabei sein dürfte.

1.5.1 Cursorsteuerung und andere einfache Editorfunktionen

Die Zeilenlänge des KICK-PASCAL-Editors ist auf 256 Zeichen begrenzt. In der Regel wird man als Programmierer nicht mehr Zeichen in eine Zeile schreiben als auf dem Bildschirm sichtbar sind (ca. 80 Stück, je nach Windowgröße), da sonst die

Übersichtlichkeit des Programms leidet. Wenn man sich nun doch mit dem Cursor dem rechten Window-Rand nähert, scrollt der Fensterinhalt um einige Zeichen nach links.

Der Editor kennt zwei Modi: Den Einfüge- und den Überschreibmodus. Mit der Tastenkombination "Ctrl-V" (in Zukunft werde ich "Ctrl-X" mit "^X" abkürzen) können Sie zwischen diesen beiden Modi unschalten. Der Unterschied liegt darin, was der Editor macht, wenn der Cursor auf einem anderen Zeichen, d. h. nicht am Ende einer Zeile oder in einer Leerzeile steht und Sie dann ein Zeichen tippen. Im Modus "Insert" wird das neue Zeichen an der aktuellen Cursorposition eingefügt, im "Overwrite"-Modus wird das Zeichen, auf dem der Cursor steht mit dem eingegebenen Zeichen überschrieben.

Auch die Wirkung der "Return"- bzw. "Enter"-Taste ist vom Modus abhängig: Im Überschreibmodus springt der Cursor ganz einfach an den Anfang der nächsten Zeile. Dagegen wird im "Insert"-Mode an der Cursorposition ein Zeilenende eingefügt. Das bedeutet, daß eine neue Zeile im Text unter der Zeile, in der der Cursor steht, eingefügt wird. Der Cursor springt nun an den Anfang der eingefügten Zeile, wobei das, was vorher rechts vom Cursor gestanden hat, in die neue Zeile übernommen wird. Außerdem werden noch Einrückungen (in PASCAL enorm wichtig!) berücksichtigt: Wenn die alte Zeile mit einigen Leerzeichen begann, werden auch an den Anfang der neuen Zeile entsprechend viele Leerzeichen gesetzt. Das klingt jetzt alles sehr verwirrend, aber Sie werden sich schnell daran gewöhnen (wenn Sie es nicht schon von anderen Editoren her gewohnt sind).

Sinn und Zweck eines jeden Editors ist, daß man seinen Text nicht wie auf einer Schreibmaschine von vorn bis hinten 'runtertippen muß, sondern jederzeit an eine andere Textstelle springen und dort etwas ändern kann. Dazu dienen zunächst einmal die vier Cursortasten: Wenn man sie "einfach so" drückt, geht der Cursor jeweils um eine Position in die gewünschte Richtung. Zusammen mit "Shift" springt die "Hoch"- Taste eine Seite nach oben (eine Seite entspricht jeweils einer Windowhöhe), die "Runter"-Taste entsprechend eine Seite nach unten. Mit der Tastenkombination "Shift-Rechts" gelangen Sie an das Zeilenende, mit "Shift-Links" an den Anfang. Eine Besonderheit gibt es, wenn die Zeile mit einigen Leerzeichen beginnt: Dann springt "Shift-Links" zunächst zum ersten Nicht-Leerzeichen der Zeile. Falls er da aber schon steht, springt er statt dessen in Spalte 1. Sie können also mit der geshifteten Cursor-Links-Taste zwischen der ersten Spalte und dem "wahren" Zeilenanfang hin- und herspringen.

Die Tasten "Links" und "Rechts" sind auch noch in Verbindung mit der "Ctrl"-Taste belegt: Damit können Sie wortweise springen. Ein "Wort" ist für den Editor eine ununterbrochene Folge von Buchstaben und/oder Ziffern. Die Taste "F1" springt an den Textanfang, "F10" ans Ende. Mit der "Tab"-Taste springt man eine Tabulatorposition nach rechts. Die Tabulatoren sind fest auf 8 Zeichen eingestellt, so daß man immer in Spalte Nummer 1, 9, 17, 25, 33 oder ... landet. Zusammen mit "Shift" " springt man entsprechend einen Tabulator nach links.

Wichtig ist auch "^#": In der Titelzeile werden Sie aufgefordert, eine Zeilennummer einzugeben. Der Cursor wird in diese Zeile gesetzt.

Für die meisten Cursorsteuer-Befehle gibt es auch Alternativen in Form von "Ctrl"-Tastenbelegungen:

^S	wie "Cursor Links"
^D	wie "Cursor Rechts"
^A	wortweise nach links (wie "Ctrl-Links")
^F	wortweise nach rechts (wie "Ctrl-Rechts")
^QS	Sprung an Zeilenanfang (wie "Shift-Links")
^QD	Sprung ans Zeilenende (wie "Shift-Rechts")
^E	wie "Cursor Hoch"
^X	wie "Cursor Runter
^R oder **^QE**	eine Seite nach oben (wie "Shift-Hoch")
^C oder **^QX**	eine Seite nach unten (wie "Shift-Runter")
^QR	Sprung an Textanfang (wie "F1")
^QC	Sprung ans Textende (wie "F10")

Wenn hier (oder im folgenden) "^AB" steht, so heißt das, daß zuerst "Ctrl-A" und danach wahlweise "Ctrl-B" oder einfach "B" zu tippen ist.

1.5.2 Löschen, Einfügen und Scrollen

Die "Del"-Taste löscht das Zeichen unter dem Cursor. "Backspace" geht erst ein Zeichen nach links und löscht dann dort ein Zeichen, dadurch rückt der Cursor samt dem Rest der Zeile um ein Zeichen nach links.

Wenn der Cursor am Zeilenanfang steht, löscht "Backspace" das Zeilenendezeichen, d. h. die aktuelle Zeile wird an die Zeile darüber angehängt.

^G	ist eine Alternative zu "Del"
^H	entspricht "Backspace"

Für größere Löschoperationen gibt es die vier folgenden Tasten:

^T	Löscht ab der Cursorposition ein Wort bzw. einen Wortzwischenraum.
^L	Alles, was rechts vom Cursor steht, wird gelöscht.
^K	Die ganze Zeile wird gelöscht, so daß eine Leerzeile zurückbleibt.
^Z	Die ganze Zeile wird völlig entfernt. Im Gegensatz zu "^K" bleibt hier nicht einmal eine Leerzeile übrig.

In einem Puffer wird das, was bei der letzten dieser vier Operationen gelöscht wurde, aufbewahrt. Mit "^U" (für "Undo") können Sie diese gepufferten Zeichen an der aktuellen Cursorposition einfügen. Dadurch ist "^U" von zweierlei Nutzen: Zum einen können Sie damit versehentliche Löschungen rückgängig machen, zum

anderen können Sie einfach kurze Textstücke verschieben und kopieren, ohne dafür einen Block markieren zu müssen (siehe Kapitel 1.5.3). Dafür müssen Sie den Text mit einer der vier genannten Tastenkombinationen löschen und dann beliebig oft an beliebig vielen verschiedenen Stellen mit "^U" wieder einfügen.
Übrigens fügt "^U" immer ein, sogar dann, wenn der Editor gerade im "Overwrite"-Modus ist.

^J	Fügt eine Leerzeile ein.

Im Editor-Fenster kann man naturgemäß nur eine begrenzte Anzahl von Textzeilen darstellen. Oft kommt es dabei vor, daß man mit dem Cursor ganz unten im Fenster steht und die nachfolgenden Zeilen sehen will, ohne den Cursor von der Stelle zu bewegen. Für solche und ähnliche Situationen gibt es die Scroll-Kommandos:

^W	Scrollt den Windowinhalt um eine Zeile nach oben.
^Y	Der Fensterinhalt wird um eine Zeile nach unten gescrollt.
^QW	Es wird so gescrollt, daß der Cursor ganz oben steht.
^QY	Entsprechend nach unten.
^QQ oder **F8**	Der Cursor wird in die Windowmitte gesetzt.

Achtung: Wenn Sie aus irgendwelchen Gründen keine deutsche Tastaturbelegung eingestellt haben, vertauschen sich "^Z" und "^Y"! Die Taste zum Löschen einer Zeile befindet sich also unabhängig von der Keymap stets zwischen dem "T" und dem "U", die Taste zum 'runterscrollen ist immer unten links neben dem "X".

1.5.3 Block-Operationen

Alle Block-Befehle werden mit der Tastenkombination "^B", gefolgt von einem weiteren Zeichen, eingeleitet. Daneben gibt es noch Alternativen in Form von Pull-Down-Menüs, zu denen es wiederum meist Hotkeys gibt.

Als erstes müssen Sie sich einen Bereich des Quelltextes als Block markieren. Sowohl Blockanfang als auch -ende werden mit "^BB" markiert, wobei die Reihenfolge der beiden Markierungen egal ist. Der Block wird dann farblich hervorgehoben. In der Editor-Titelzeile wird die Markierung durch eckige Klammern bestätigt: Eine einzelne Klammer "[", wenn erst eine Grenze markiert wurde, ein Klammernpaar "[]" nach dem nächsten "^BB".

Man kann aber auch das Pull-Down-Menü "Editor/Block/Markieren" bzw. die Hotkey-Kombination "rechte AMIGA-Taste"+"B" benutzen.
Die Markierung eines nicht mehr benötigten Blocks kann mit "^BF" (für "Forget") aufgehoben werden.

Mit "^BS" können Sie den Cursor an den Anfang eines markierten Blocks setzen, "^BE" springt entsprechend an das Blockende.

Die wichtigsten Blockoperationen sind:

^BC	Block an Cursorposition kopieren
^BV	Block an Cursorposition verschieben
^BD	Block (nach Sicherheitsabfrage) löschen. Dafür gibt es auch Menü-Alternativen im Untermenü "Editor/Block".

Block-Datei-Befehle:

^BW	Der markierte Block wird in eine Datei geschrieben. In der Editor-Kopfzeile wird nach dem Dateinamen gefragt. Das Menü "Editor/Datei/ Block schreiben" macht im Prinzip dasselbe, dort wird aber ein Filerequester benutzt.
^BD	Eine Textdatei wird an der aktuellen Cursorposition in den Text eingefügt und als Block markiert.
^BP	Der Block wird gedruckt.

Gerade als PASCAL-Programmierer steht man oft vor dem Problem, daß man einen Bereich des Quelltextes seitlich verschieben will, z. B. wenn man um eine längere Anweisungsfolge eine zusätzliche "BEGIN"- "END"-Klammer gelegt hat. Auch hier läßt Sie der KICK-PASCAL-Editor nicht im Stich: Mit der Kombination "^B"+"Cursor links" bzw. "rechts" wird ein Block horizontal verschoben, d. h. ein Zeichen weiter bzw. weniger weit eingerückt. Da diese Tastenbelegungen auf Dauer zu beidseitigen Handkrämpfen führt, gibt es auch hier mehrere andere Möglichkeiten: Zum einen das Untermenü "Editor/Block schieben" und zum anderen die Tasten "F5" und "F6".

1.5.4 Suchen und Ersetzen

Der KICK-PASCAL-Editor kann Stellen im Text suchen. Dazu gibt es wieder einmal zwei Möglichkeiten: erstens die Tastenkombination "^QF" und zweitens das Menü "Editor/Finden" bzw. entsprechende Hotkey-Tastenkombination "AMIGA-F". Im ersten Fall werden Sie zunächst in der Editor-Kopfzeile nach dem zu suchenden Text gefragt, anschließend erscheint dort ein Tastaturmenü mit den verschiedenen Optionen, die Sie mit ihren Anfangsbuchstaben an- und ausschalten und mit "Return" bestätigen können .

Falls Sie das Pull-Down-Menü benutzen, erscheint ein Requester. Das Text-Gadget zur Eingabe des Suchstrings ist bereits aktiviert, so daß Sie gleich lostippen können. Sollte im Textfeld noch eine ältere Eintragung stehen, kann man sie mit "AMIGA-X" löschen. Tippen Sie nun den Suchtext und bestätigen Sie wie immer mit "Return". Unter dem Textfeld sehen Sie die Gadgets für die vier Suchoptionen. Wenn die Einstellung der Schalter O.K. ist ("rot" bedeutet eingeschaltet), können Sie wahlweise die Gadgets "OK"/"Abbruch" oder die Tasten "Return"/"Esc" benutzen. Sie können diesen Requester also völlig ohne Maus benutzen!

Nun aber zu den bereits erwähnten Such-Optionen:

- "Nur Worte" / "Word":
  Es sollen ausschließlich Worte gesucht werden. Eine Zeichenfolge gilt hier als Wort, wenn im Text weder vor noch hinter ihr ein Buchstabe oder eine Ziffer stehen.
- "Vorwärts" / "Forward":
  Der Text ist von der Cursorposition oder vom Textanfang (je nach Einstellung des "Ab Cursor"-Switchs) bis zum Textende zu durchsuchen. Ist diese Option ausgeschaltet, so wird von der Cursorposition bzw. vom Textende rückwärts bis zum Textanfang gesucht.
- "IgNoRiErEn" / "Upcase":
  Es soll nicht zwischen Groß- und Kleinbuchstaben unterschieden werden.
- "Ab Cursor" / "Cursor":
  Es soll an der Cursorposition mit dem Suchen angefangen werden, andernfalls am Anfang oder Ende des Quelltextes (je nach Wahl der "Vorwärts"-Option).

Der Cursor wird dann an die Position des ersten gefundenen Auftretens des Suchtextes gesetzt. Wenn Sie weitersuchen lassen wollen, können Sie mit dem Menü "Editor/Wiederhole" bzw. dem Hotkey "AMIGA-W" die letzte Suchoperation wiederholen lassen. Die Option "Ab Cursor" wird dabei automatisch eingeschaltet, denn sonst würde ja immer wieder dieselbe Position gefunden werden.

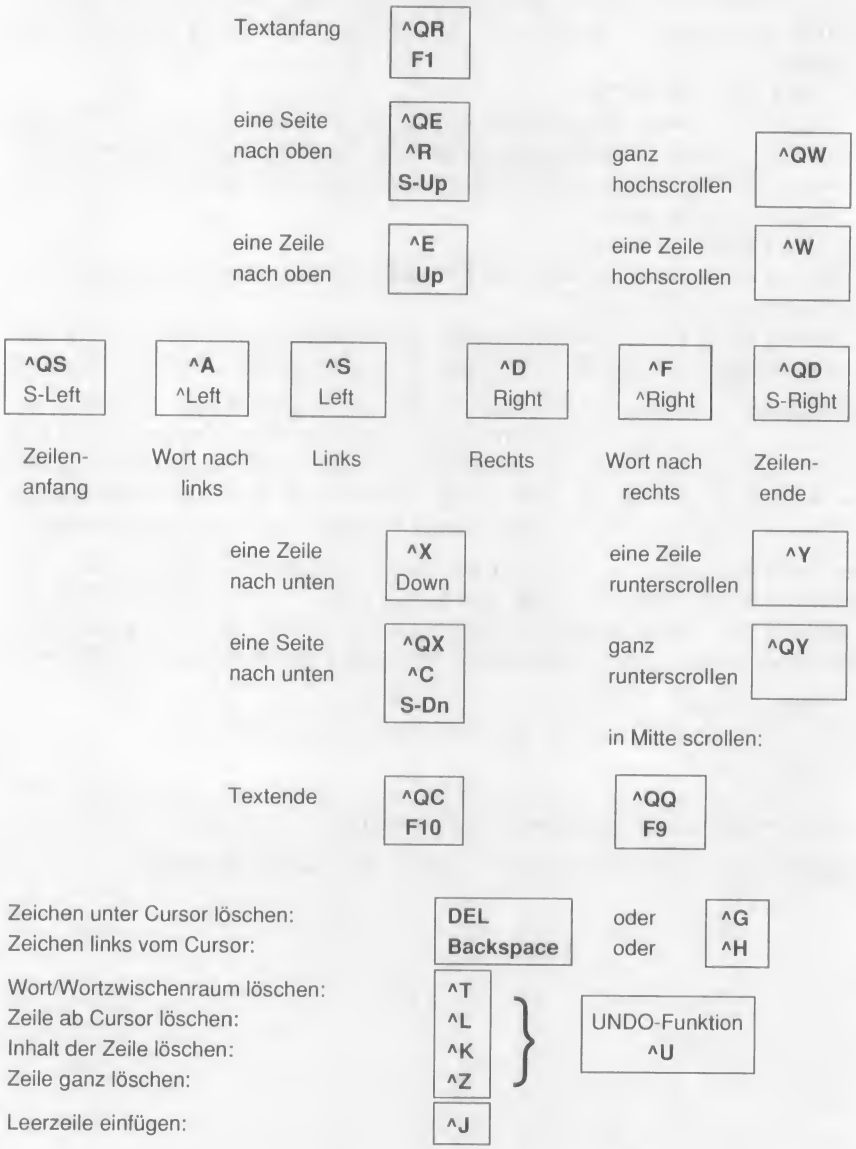
Um eine Zeichenfolge zu suchen und automatisch durch eine andere zu ersetzen, gibt es das Menü "Editor/Ersetzen" und die Tastenkombination "^QA". Die Bedienung entspricht der des Such-Befehls, aber hier ist natürlich noch ein zweiter String einzugeben (mit dem ersetzt werden soll). Außerdem gibt es zwei weitere Optionen:

- "Zuerst fragen" / "Question"
  Vor jeder Ersetz-Operation ist sicherheitshalber nachzufragen.
- "Alle" / "All"
  Der Suchstring ist an jeder Textstelle, an der er gefunden wird, zu ersetzen, andernfalls nur an der ersten gefundenen Stelle.

Auch die Ersetz-Funktion kann mit "AMIGA-W" wiederholt werden.

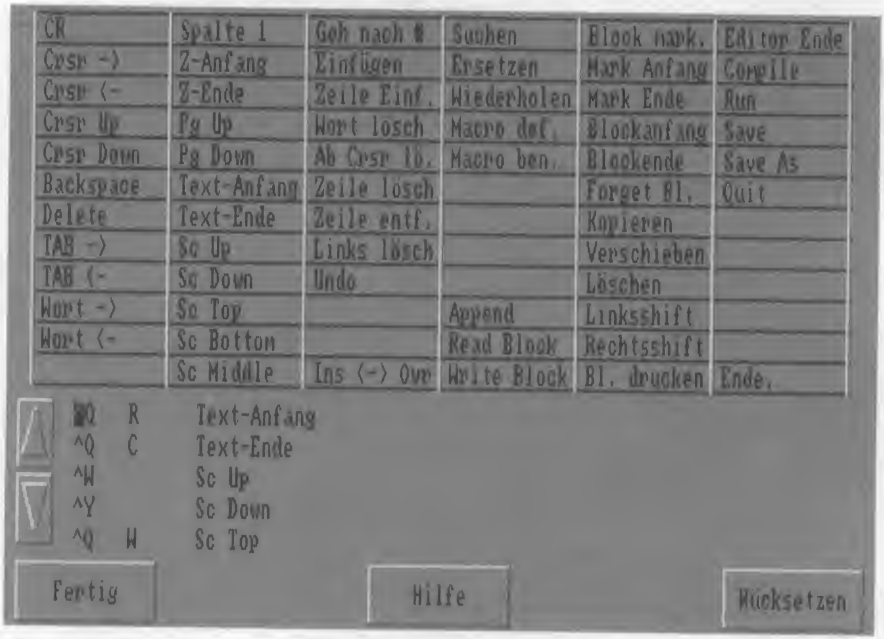
1.5.5 Kurzübersicht

Cursorsteuerung und Scrolling:



Block-Operationen	
-markieren:	**^BB** oder AMIGA-**B**
-Markierung aufheben:	**^BF**
-kopieren:	**^BC** oder AMIGA-**C**
-verschieben:	**^BV** oder AMIGA-**V**
-löschen:	**^BD** oder AMIGA-**D**
-in Datei schreiben:	**^BW**
-aus Datei lesen:	**^BR**
-drucken:	**^BP**
-Cursor an Anfang:	**^BS**
-Cursor an Ende:	**^BE**
-nach links schieben:	**^B**-Cursor links oder **F5**
-nach rechts schieben:	**^B**-Cursor rechts oder **F6**
Suchen:	**^QF** oder AMIGA-**F**
Suchen und ersetzen:	**^QA** oder AMIGA-**R**
Letzte Such-/Ersetzoperation wiederholen:	AMIGA-**W**

1.6 Das Programm PASCALPrefs



Bei der Entwicklung des KICK-PASCAL-Systems war es ein Ziel, eine möglichst große Flexibilität zu erreichen. Der Anwender sollte die Möglichkeit erhalten, das System weitgehend an seine Bedürfnisse anzupassen. Deshalb benötigt KICK-PASCAL seit Version 2.0 eine Konfigurationsdatei, die unter dem Namen "Pascal.config" im "S:"-Verzeichnis liegen sollte (beim Start aus dem CLI können Sie aber auch einen anderen Dateinamen angeben).

In dieser Datei stehen nun die unterschiedlichsten Daten. Sie ist eine ASCII-Datei, kann also im Prinzip auch mit einem Editor bearbeitet werden. Allerdings sind die meisten Einträge nicht ohne weiteres verständlich. Deshalb gibt es ein sehr komfortables Programm, mit dem Sie KICK-PASCAL-Konfigurationsdateien erstellen und bearbeiten können: Es heißt "PascalPrefs" und befindet sich auf Diskette 1.

Nach dem Programmstart (von Workbench oder CLI) werden Sie zunächst nach dem Namen der Config-Datei gefragt. Der Standardname "S:Pascal.config" ist vorgegeben und kann bestätigt oder geändert werden. Falls keine Datei unter dem angegebenen Namen existiert, fragt das Programm, ob eine entsprechende Datei erzeugt werden soll. Dabei werden zunächst die Standard-Einstellungen benutzt.

Nun gelangen Sie in das Hauptmenü von "PascalPrefs". Ganz oben sehen Sie ein Textgadget mit dem Dateinamen, darunter die einzelnen Menüpunkte, die Sie mit der Maus anklicken können. Im einzelnen handelt es sich dabei um die folgenden Programmfunktionen:

Laden

Eine Konfigurationsdatei wird geladen. Als Dateiname wird der Inhalt des entsprechenden Textgadgets benutzt. Falls Sie zuvor schon etwas an der alten Einstellung geändert haben, gehen diese Änderungen beim Laden einer anderen Konfiguration natürlich verloren.

Abspeichern

Die Einstellungen werden in einer Konfigurationsdatei abgespeichert.

Editor-Tasten belegen

Es ist ein besonders interessantes Feature von KICKPASCAL 2.0, daß man die Tastenbelegung des Editors weitgehend frei wählen kann. Dies ist vor allem dann angenehm, wenn man noch einen anderen Editor benutzt und sich nicht ständig umgewöhnen will.

Der Bildschirm des "Editor-Editors" ist zweigeteilt: Oben sehen Sie eine Tabelle mit den verschiedenen Editorfunktionen, unten einen fünf Zeilen langen Ausschnitt aus der Tastaturbelegungsliste. Dort steht ein Textcursor, den Sie mit den Cursortasten durch die Liste bewegen können. Mit den beiden Pfeilgadgets links davon können Sie schneller durch die Liste springen.

Ein Eintrag in der Liste besteht immer aus einer oder zwei Tasten und bis zu sechs Funktionen, die beim Drücken der Taste bzw. Tastenfolge nacheinander ausgeführt werden sollen.

Die wichtigsten Tastenfunktionen, nämlich die Cursortasten sowie "Return", "Tab", "Delete" und "Backspace", lassen sich nicht verändern. Sie können nur Tasten bzw. Folgen von zwei Tasten eine Folge von Editorfunktionen zuordnen. Dabei ist die erste Taste immer "Escape", eine Funktionstaste oder eine Tastenkombination mit "Ctrl".

Als zweite Taste, falls vorhanden, ist jede andere Taste außer "Space" und "Return" möglich. Dabei wird nicht zwischen Groß- und Kleinbuchstaben unterschieden. Mit den Tasten "D" oder "Del" können Sie die Zeile, auf der der Cursor steht, aus der Liste löschen. Mit "I" können Sie eine neue Zeile einfügen. Man ändert eine Tastenkombination (also die linke Spalte eines Listeneintrags), indem man den Cursor darauf setzt und einfach die beiden Tasten eintippt. Wenn die zweite Taste entfallen soll, gibt man hier "Space" oder "Return" ein.

Die Folge der Tastenfunktionen kann man wahlweise mit der Maus oder mit der Tastatur eingeben. Zunächst ist auch hier der Textcursor auf den zu ändernden Eintrag in der rechten Spalte zu setzen. Dann kann man mit der Maus bis zu sechs Funktionen aus der oberen Bildschirmhälfte auswählen und mit einem Klick auf "Ende." beenden.

Alternativ können Sie auch "Space"drücken. Nun erscheint in der Funktionentabelle ein hervorgehobenes Feld. Sie können diesen Cursor dann mit den Pfeiltasten durch

die Tabelle bewegen, mit "Space" Funkionen auswählen und mit "Return" beenden.

Ein Beispiel: Sie wollen zwei neue Tastenfuntionen in die Tabelle aufnehmen. Zum einen soll "F2-X" den Text abspeichern und danach das KICK-PASCAL-Systen verlassen, zum anderen soll "F3" als Alternative zu "^B-F" die Markierung eines Blocks aufheben.

Als erstes setzen Sie den Cursor an eine geeignete Stelle der Liste und fügen mit "I" eine neue Zeile ein (sonst würden Sie einen alten Eintrag überschreiben), oder Sie setzen ihn ans Ende der Liste.

Auf jeden Fall sollte er aber in der linken Spalte stehen, da Sie sinnvollerweise als zuerst die Tasten eingeben sollten. Nun drücken Sie nacheinender die Tasten "F2" und "X". Der Cursor sollte dann in der rechten Spalte stehen. Mit der Maus klicken Sie nun nacheinander "Save", "Quit" und "Ende" an, und schon steht eine neue Zeile in der Liste. Jetzt noch die zweite neue Funktion: Setzen Sie den Cursor wieder an das Listenende oder fügen Sie irgendwo eine neue Zeile ein.

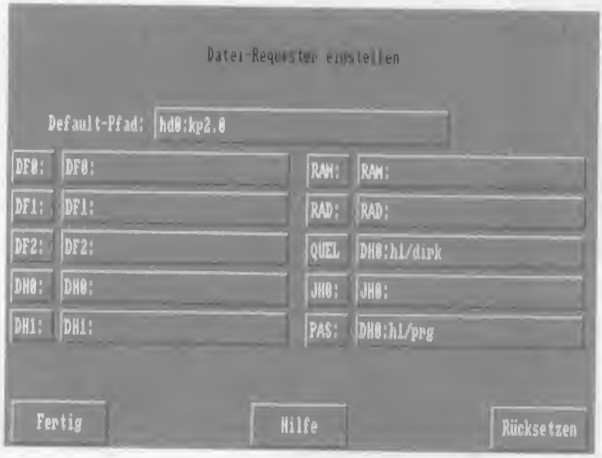
Drücken Sie zuerst "F3" und dann "Space" oder "Return", denn Sie wollen diesmal ja eine einzelne Taste belegen. Jetzt steht der Cursor wieder in der rechten Spalte, aber zur Abwechslung wählen wir die Funktion diesmal mit der Tastatur aus (das geht mit etwas Übung auch schneller, da man nicht ständig von der Tastatur zur Maus und zurück greifen muß). Drücken Sie also "Space", bewegen Sie das hervorgehobene Feld mit den Cursortasten auf "Forget Bl." und tippen Sie "Space" (zum Anwählen der Funktion) und "Return" (um die Eingabe zu beenden).

Die folgende Liste gibt an, was die aus Platzgründen teilweise arg abgekürzten Funktionsbezeichnungen bedeuten. Bei den meisten ist nur angegeben, welcher Taste in der Standard-Tastenbelegung die Funktion entspricht; bei den Funktionen, die in der Standard-Belegung nicht benutzt werden, finden Sie eine kurze Erläuterung.

CR	= Return
Crsr ->	= Cursor rechts oder **^D**
Crsr <-	= Cursor links oder **^L**
Crsr Up	= Cursor nach oben oder **^E**
Crsr Down	= Cursor nach unten oder **^X**
Backspace	= Backspace
Delete	= Delete oder **^G**
TAB ->	= Tab
TAB <-	= Shift-Tab
Wort ->	= Ctrl-Cursor rechts oder **^F**
Wort <-	= Ctrl-Cursor links oder **^A**
Spalte 1	Cursor in erste Spalte setzen
Z-Anfang	= Shift-Cursor links oder **^QS**
Z-Ende	= Shift-Cursor rechts oder **^QD**
Pg Up	= Shift-Cursor nach oben oder **^R**

Pg Down	= Shift-Cursor nach unten oder **^C**
Text-Anfang	= **F1** oder **^QR**
Text-Ende	= **F10** oder **^QC**
Sc Up	= **^W**
Sc Down	= **^Y**
Sc Top	= **^QW**
Sc Bottom	= **^QY**
Sc Middle	= **^QQ** oder **F8**
Geh nach #	= **^#**
Einfügen	an Cursorposition ein Leerzeichen einfügen
Zeile einf.	= **^J**
Wort lösch	= **^T**
Ab Crsr lö.	= **^L**
Zeile lösch	= **^K**
Zeile entf.	= **^Z**
Links lösch	Zeileninhalt links von Cursor löschen
Undo	= **^U**
Ins <-> Ovr	= **^V**
Suchen	= **^QF**
Ersetzen	= **^QA**
Wiederholen	= Amiga-**W**
Append	Datei an Text anhängen
Read Block	= **^BR**
Write Block	= **^BW**
Block mark.	= **^BB**
Mark Anfang	Blockanfang markieren oder ändern
Mark Ende	Blockende markieren oder ändern
Blockanfang	= **^BS**
Blockende	= **^BE**
Forget Bl.	= **^BF**
Kopieren	= **^BC**
Verschieben	= **^BV**
Löschen	= **^BD**
Linksshift	= **F5**
Rechtsshift	= **F6**
Bl. drucken	= **^BP**
Editor Ende	= **Esc**
Compile	= Shift-**F9**
Run	= **F9**
Save	Text abspeichern
Save as	Text unter neuem Namen abspeichern
Quit	KICK-PASCAL verlassen

File-Requester einstellen



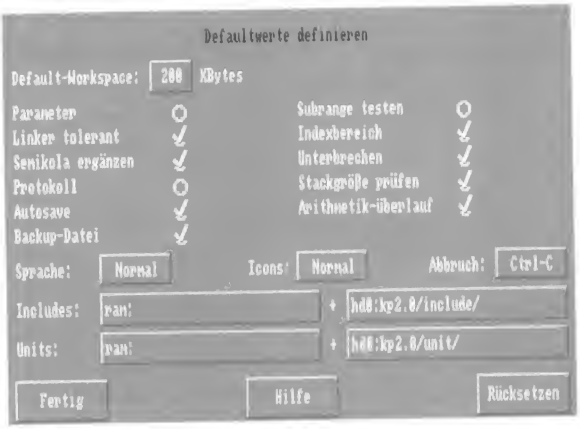
Auch für die Dateiauswahlbox von KICK-PASCAL gibt es verschiedene Einstellmöglichkeiten. Zunächst wäre da der "Default-Pfad", das ist das Verzeichnis, welches bei der ersten Benutzung des Filerequesters eingelesen werden soll. Standardeinstellung ist "SYS:". Wenn Sie hier nichts eingeben, wird das aktuelle Directory verwendet.

Als Festplattenbenutzer können Sie hier das Directory angeben, in dem Sie Ihre Pascal-Quelltexte normalerweise ablegen. Wenn Sie keine Harddisk besitzen, werden Sie wegen der begrenzten Diskettengröße auf Dauer nicht mit nur einem Quelltextverzeichnis auskommen. Dann bietet es sich an, hier eine Laufwerksbezeichnung ("DF0:" oder so) oder nichts (also das aktuelle Verzeichnis) anzugeben.

Die zehn Gadgets darunter entsprechen den zehn Geräteknöpfen des Requesters. Eine kleine Besonderheit ist, daß die Beschriftung des Knopfs (maximal 4 Zeichen lang) und der tatsächliche Pfad, mit dem der Knopf belegt ist (bis zu 40 Zeichen) nicht übereinstimmen müssen.
Sie können die Knöpfe also nicht nur mit Geräten, sondern auch mit Unterverzeichnissen belegen.

Bitte achten Sie aber darauf, daß Sie weder am Anfang noch am Ende eines Pfadnamens ein Leerzeichen eingeben, denn das würde als Teil des Namens betrachtet.

Defaultwerte definieren



Im "Defaultwerte"-Bildschirm können Sie im wesentlichen die Startwerte für die Einstellungen, die man im Pull-Down-Menü "Optionen" vornehmen kann, definieren. Sie müssen dann nicht mehr bei jedem Start des KICK-PASCAL-Systems Ihre bevorzugten Einstellungen wählen. "Default-Workspace" ist die Arbeitsspeichergröße, die beim Start des KICK-PASCAL-Systems vorgegeben wird (Standard: 80 kByte).

Darunter finden Sie 11 abhakbare Punkte, die folgenden Einstellmöglichkeiten in den Pull-Down-Menüs entsprechen:

PascalPrefs-Option	entsprechendes Menü
Parameter	Starten/Parameter
Linker tolerant	Optionen/Linker
Semikola ergänzen	Optionen/Spezial/Ergänze Semikola
Protokoll	Optionen/Protokoll
Autosave	Optionen/Dateien/Autosave
Backup-Datei	Optionen/Dateien/Backup
Subrange testen	Optionen/Compiler/Subrange testen
Indexbereich	Optionen/Compiler/Indexbereich
Unterbrechen	Optionen/Compiler/Unterbrechen
Stackgröße prüfen	Optionen/Compiler/Stackgröße
Arithmetik-Überlauf	Optionen/Compiler/Arithm. Überl.

Ferner können Sie einstellen, ob Icons erzeugt werden sollen. Dabei gibt es drei verschiedene Einstellungen: "Ein", "Aus" und "Normal". Letzteres heißt, daß das Menü "Optionen/Dateien/Icons" nur beim Start des KICK-PASCAL-Systems von der Workbench angewählt ist.

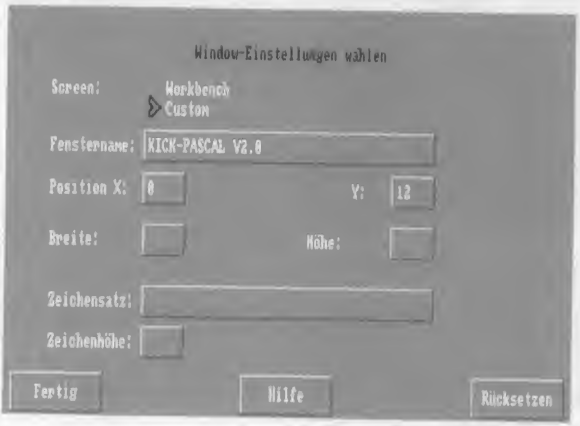
Auch bei der Wahl der Sprache gibt es drei Möglichkeiten: "Deutsch", "Englisch" und "Normal", d. h. Sie können entweder eine Sprache für die KICK-PASCAL-Benutzeroberfläche festlegen oder es wie gewohnt von der Tastaturbelegung abhängig machen.

Zu guter Letzt können Sie noch die Verzeichnisse angeben, in denen Include-

Dateien bzw. Units gesucht werden sollen. Diese Einstellungen entsprechen denen im "Suchpfade"-Requester des KICK-PASCAL-Systems.

Dabei haben Sie wie immer die Möglichkeit, jeweils zwei Verzeichnisse anzugeben, wobei Dateien, die nur im zweiten Verzeichnis gefunden werden, automatisch in das erste umkopiert werden.

Window-Einstellungen



Mit dem vorletzten Punkt des Hauptmenüs können Sie das PASCAL-Window Ihren Bedürfnisse anpassen. Dies wurde vor allem durch die zahlreichen neuen Grafikmodi des AMIGA 3000 nötig.

Zunächst stehen Sie vor der Entscheidung, ob Sie ein PASCAL-Window auf dem Workbench-Screen bevorzugen ("Workbench") oder ob KICK-PASCAL einen eigenen Screen öffnen soll ("Custom"). Beides hat seine Vor- und Nachteile und wurde deshalb dem individuellen Geschmack des Benutzers überlassen, ebenso wie der Titel des Windows bzw. Screens, den Sie im zweiten Gadget eingeben können.
Es folgen vier Gadgets, in denen Position und Größe des Windows bzw. Screens einzugeben sind. Wenn man bei "Breite" oder "Höhe" nichts einträgt, werden hier die entsprechenden Abmessungen des Workbenchscreens benutzt. Die beiden "Position"-Angaben haben keine Bedeutung, wenn Sie einen eigenen Screen öffnen.

Als letztes können Sie optional noch einen besonderen Zeichensatz für den Editor wählen. Er wird bestimmt durch den Namen (z. B. "topaz") und die Zeichenhöhe. Es ist nicht empfehlenswert, hier einen Proportionalschrift-Font anzugeben, denn damit kommt der Editor nicht klar.

Programmende

Mit einem Klick auf "Programmende" verlassen Sie "PascalPrefs". Bitte achten Sie darauf, daß Sie Ihre geänderten Einstellungen vorher abgespeichert haben, denn sonst gehen sie verloren.

1.7 Details für Fortgeschrittene

1.7.1 Belegung des Arbeitsspeichers

Zu jeder passenden Gelegenheit gibt KICK-PASCAL die genaue Belegung des reservierten Arbeitsspeichers aus. Man kann sie sich auch im Tastaturmenü mit der Leertaste ausgeben lassen. Dabei werden Start- und Endadresse der Blöcke jeweils hexadezimal und ihre Länge dezimal ausgegeben.

Die erste Zeile ("Text:") enthält immer die Länge des Textes im Editor. Sie ist aber mehr von statistischem Interesse. Wichtiger ist da schon "Free": hier steht, wieviel Arbeitsspeicher noch frei ist.

Wenn das Programm in compilierter Form vorliegt, enthält die Liste noch einige weitere Einträge: Zwischen den Zeilen "Text" und "Free" stehen dann noch Größe und Adresse des erzeugten Programmcodes. Außerdem stehen darunter noch Angaben über "Reloc", die Relokationstabelle. Sie liegt nicht im reservierten Arbeitssppeicher und enthält Linkerdaten sowie Informationen darüber, wie das Programm zu modifizieren ist, wenn es in einen anderen Speicherbereich verschoben wird (wichtig für die Erzeugung von Exe-Files).

Die letzte Zeile ist normalerweise:

```
Linear stack requirenent = xxxx ($xxxx) bytes.
```

Dies gibt eine Abschätzung darüber, wieviel Stack das Programm zur Laufzeit benötigt. "Linear" bedeutet, daß dabei Rekursionen nicht berücksichtigt werden. Zusätzlich braucht man noch ca. 1 KByte, um Betriebssystem-Funktionen aufrufen zu können. Wenn das Programm aus der KICK-PASCAL-Entwicklungsumgebung gestartet wird, benutzt es den noch freien Arbeistspeicher als Stack. Wird es dagegen vom CLI gestartet, so müssen Sie dafür Sorge tragen, daß genügend Stack reserviert ist (mit dem gleichnamigen CLI-Befehl). Beim Workbenchstart enthält das dazu nötige Icon einen entsprechenden Eintrag. Mit dem Workbench-Menüpunkt "Info" kann man diesen Wert ändern.

Wenn das Programm nicht nur compiliert, sondern auch korrekt gelinkt wurde, wird auch noch eine Zeile der Art

```
Linker Complete - Maximum code size = xxxx ($xxxx) Bytes
```

ausgegeben. Dabei fällt auf, daß der Wert um einige KBytes größer als der reine Programmcode ist, denn hier wird die Codelänge der Exe-Datei angegeben, die auch die automatisch hinzugelinkten Teile des PASCAL-Laufzeitsystems enthält. Die Exe-Datei selbst ist übrigens noch ein paar Bytes länger, denn das AMIGA-Programmformat enthält noch einige weitere Informationen (z. B. die oben bereits erwähnte Relokationstabelle). Das Wort "Maximum" hat übrigens noch keine Bedeutung. Prinzipiell kann der AMIGA nämlich auch mit Overlays arbeiten, wodurch die Codelänge schwanken kann. Der KICK-PASCAL Linker unterstützt aber in der vorliegenden Version noch keine Overlays.

Wenn Sie mit Modulen oder statischen Variablen arbeiten, enthält die Speicherbelegungstabelle noch einen oder mehrere mit "Link" betitelte Einträge. Dies ist eine Liste der "Hunks" des Gesamtprogramms.

Nach einem Laufzeitfehler oder einem gewaltsamen Programmabbruch ("F10") lautet die unterste Zeile der Speicherbelegung:

```
Program terminated at $xxxxxx.
```

Dies sagt Ihnen, daß Sie die Funktion "Find Error" bzw. "Fehlersuche" benutzen können, um die Quelltextposition des Programmabbruchs suchen zu lassen.

1.7.2 Der Exception-Handler

Eine Exception ist im Prinzip so etwas wie ein Laufzeitfehler des Prozessors. Während sie normalerweise zu einer Guru Meditation führen, fängt das KICK-PASCAL-System solche "harmlosen" Abstürze ab. Dabei werden Codeadresse und Fehlername ausgegeben.

Typische Situationen, die in einem PASCAL-Programm eine Exception auslösen können, sind z. B., wenn man vergessen hat, einem Pointer einen sinnvollen Wert zu geben oder eine Library zu öffnen. In der Regel kann man nach einer Exception normal weiter arbeiten. Allerdings kann es vorkommen, daß das PASCAL-Programm vor der Exception bereits Amok gelaufen ist und Daten des Betriebssystems beschädigt hat. Daher sollten Sie den Hinweis "System may be corrupted", der bei den schwerwiegenderen Fehlern ausgegeben wird, ernst nehmen. Wenn das PASCAL-Programm mit einer Exception aussteigt, werden auch nicht - wie bei den normalen Laufzeitfehlern - Windows und Dateien geschlossen oder reservierter Speicher freigegeben. Deshalb ist es oft besser, nach einem solchen Beinahe-Absturz den Quelltext abzuspeichern und neu zu booten.

Der Exception-Handler ist Bestandteil der Entwicklungsumgebung, nicht des PASCAL-Laufzeitsystems. Deshalb werden derartige Fehler in Exe-Files nicht abgefangen - hier führen sie zum beliebten Requester "Software Error - Task held. Finish ALL Disk activity.".

Clevererweise fängt der Exception-Handler nicht nur Aufhänger von PASCAL-Programmen, sondern auch solche des KICK-PASCAL-Systems selbst ab. Natürlich wurde größte Sorgfalt darauf verwandt, daß KICK-PASCAL nicht "einfach so" abstürzt. Sollte aber trotzdem einmal der Compiler eine Exception verursachen, erhalten Sie die Fehlermeldung "Double panic - General protection". Falls die Entwicklungsumgebung sich aufhängen sollte, versucht das Programm noch, sich sauber aus dem System zu entfernen. Dabei werden Sie wie gewohnt gefragt, ob Sie den Quelltext abspeichern wollen. In diesem Fall hat es keinen Zweck, den Knopf "zurück" anzuklicken - Sie sind am Point of no Return.

REFERENZ

KAPITEL II

SYNTAX

II. Syntax

2.1 Allgemeines

2.1.1 Syntaktische Notation

In diesem Handbuch wird als Notation für formale Syntaxdefinitionen EBNF, Niklaus Wirths "Extended Backus Naur Form", verwendet:

"Abc"	Der Text zwischen den Anführungszeichen ist so zu übernehmen.
Abc	Entweder ist hier eine andere Syntax dieses Namens einzusetzen, oder im nachfolgenden Text steht eine Erläuterung zu diesem Punkt.
{ ... }	Der Inhalt der Klammer kann beliebig oft, inklusive Null-mal, gesetzt werden.
[...]	Der Inhalt der Klammern ist optional, d. h. er kann entweder genau einmal gesetzt oder ersatzlos weggelassen werden.
(...)	Runde Klammern dienen zur Gliederung und Strukturiereung des Ausdrucks.
A | B	Alternative: Entweder A oder B

Ein Beispiel:

```
"(" Zahl { "," Zahl } ")"
```

Dies ist eine geklammerte Liste von wie auch immer gearteten Zahlen: Am Anfang muß immer eine runde Klammer "(" stehen, gefolgt von einer ersten Zahl. Danach kann man beliebig oft jeweils ein Komma "," und eine Zahl schreiben, bevor man am Ende wieder eine Klammer ")" setzt.

Folgende Zeilen würden also der Syntax genügen:

```
(17,4)
(0)
(1, 2 , 3 , 1,-5)
```

Wie Sie sehen, dürfen Sie auch Leerzeichen einschieben, obwohl diese nicht in der formalen Syntax enthalten sind. Es gibt oft auch den umgekehrten Fall, wie bei der follgenden Syntax (die übrigens kein sinnvoller Teil von Pascal ist):

```
"BEGIN" (Bezeichner | Zahl) "END"
```

Hier ist zwischen "BEGIN" und "END" wahlweise ein Bezeichner oder eine Zahl zu setzen. Hier MUSS man sogar zusätzliche Leerzeichen einfügen:

```
BEGIN42END
```

entspräche nicht der Syntax, da dies für den Compiler ein einziger langer Bezeichner wäre.

Wie ist dieses Problem formal zu lösen? Schließlich wollen wir nicht ständig in Syntaxnotationen Leerzeichen und sonstige Trennungen einfügen.

Der Compiler analysiert den Quelltext auf zwei Ebenen: der lexikalischen und der syntaktischen. Auf der lexikalischen Ebene wird der Text in eine unstrukturierte Folge von Grundsymbolen unterteilt.

Solche Grundsymbole sind im wesentlichen Bezeichner, Wortsymbole, Integerzahlen, Gleitpunktkonstanten, Stringkonstanten und spezielle Symbole wie z. B. das Semikolon oder der Doppelpunkt. So würde der Compiler bei der lexikalischen Analyse "BEGIN42END" als ein einziges Grundsymbol, nämlich einen Bezeichner, erkennen. Um daraus mehrere zu machen, muß man sie irgendwie trennen. In Pascal dürfen zwischen Grundsymbolen beliebig viele Leerzeichen, Zeilenenden, Kommentare usw. stehen, ohne daß der Sinn des Textes dadurch irgendwie beeinflußt wird.

```
BEGIN 42END
```

entspräche unserer Beispiel-Syntax: "BEGIN" ist als erstes Symbol klar zu erkennen. Es folgt eine Zahl, die zwangsläufig da endet, wo keine Ziffer mehr folgt. Somit ist auch das Symbol "END" klar als solches zu identifizieren. Jetzt erst wird die syntaktische Analyse durchgeführt, die die Grundsymbolfolge als korrekt erkennt. Wenn nicht ausdrücklich anders angegeben, beschreiben EBNF-Ausdrücke in diesem Handbuch Regeln für die syntaktische Analyse, machen also Aussagen über die Abfolge von PASCAL-Grundsymbolen. Dadurch ist also z. B. bei unserer Beispielsyntax

```
"BEGIN" (Bezeichner | Zahl) "END"
```

inhärent klar, daß das "BEGIN" vom nachfolgenden Grundsymbol - hier wahlweise eine Zahl oder ein Bezeichner - getrennt sein muß, und daß andererseits zwischen den Symbolen beliebig viele Leerzeichen, Zeilenenden, Kommentare o. ä. gesetzt werden dürfen. Des weiteren werde ich in Zukunft nicht mehr darauf hinweisen, daß in PASCAL die Groß-Kleinschreibung keine Rolle spielt.

2.1.2 Symbole

Wie oben bereits angedeutet, gibt es in PASCAL Wort- und Spezialsymbole. Zunächst eine Liste der Wortsymbole:

AND	ARRAY	BEGIN	CASE
CONST	DIV	DO	DOWNTO
ELSE	END	FILE	FOR
FUNCTION	GOTO	IF	IN
LABEL	MOD	NIL	NOT

OF	OR	PROCEDURE	PROGRAM
PACKED	RECORD	REPEAT	SET
SHL	SHR	THEN	TO
TYPE	UNTIL	VAR	WHILE
WITH	XOR		

KICK-PASCAL wurde gegenüber dem Standard stark erweitert, was es erforderlich machte, viele Wortsymbole zu ergänzen. Nun können Wortsymbole nicht überdefiniert werden, wodurch es dadurch zu Kompatiblitätsproblemen kommen könnte. Deshalb wurden nur drei "echte" Wortsymbole hinzugenommen, nämlich XOR, SHL und SHR. Alle anderen wurden als sog. Directives realisiert. Sie unterscheiden sich von Wortsymbolen dadurch, daß sie "überdefiniert" werden können. Hier ist eine Liste dieser Directives. Der Jensen-Wirth-Standard kennt übrigens nur eine einzige Directive, nämlich FORWARD.

ABSOLUTE	EXPORT	EXTERNAL	FORWARD
FROM	IMPLEMENTATION	IMPORT	INTERFACE
LIBRARY	MODULE	OTHERWISE	STATIC
STRING	UNIT	USES	

Außerdem gibt es noch Spezialsymbole. Sie bestehen stets aus einem oder zwei Zeichen, die keine Buchstaben sind. Die folgende Liste, die übrigens genau dem Standard entspricht, enthält alle Spezialsymbole:

```
(   )   *   +   ,   -.   ..   /   :   :=   ;
<   <=   <>   =   >   >=
[   ]   ^
```

Ersatzschreibweisen:
(. für [
.) für]
@ als Ersatz für ^ (Jensen-Wirth) ist nicht möglich.

2.1.3 Das Semikolon - das ungeliebte Trennzeichen

Viele genervte Programmierer wird es freuen, daß KICK-PASCAL so gut wie keine Semikola ";" erwartet. Prinzipiell gilt, daß man sie nur da setzen muß, wo sie als Trennzeichen nötig sind. Man sollte sie aber trotzdem setzen (zumal es der "echte" PASCAL-Freak schon fast im Unterbewußtsein tut): Zum einen kann der Compiler dann die Fehler leichter analysieren und zum anderen bleiben die Programme dadurch kompatibel und portabel. Zudem kann und will ich nicht garantieren, daß der Compiler ohne Semikola 100%-ig richtig arbeitet (soll heißen, daß er eventuell hier oder da Fehler melden könnte(!), die keine sind - die Codeerzeugung dürfte dadurch unberührt bleiben).

Es gibt aber auch die Möglichkeit, im Quelltext fehlende Semikola automatisch ergänzen zu lassen.

2.1.4 Kommentare

Kommentare können wahlweise mit "{" oder "(*" beginnen und mit "}" oder "*)" enden. Geschachtelte Komentare sind möglich.

2.1.5 Bezeichner

Bezeichner ("Identifier") dienen in PASCAL dazu, Variablen, Prozeduren und vielen anderen Dingen mehr Namen zu geben. Erlaubte Zeichen in Bezeichnern sind: - Buchstaben - Ziffern - der Unterstrich "_" Bei KICK-PASCAL zählen die Umlaute "ä", "ö", "ü" zu den Buchstaben, nicht jedoch das "ß". Das erste Zeichen eines Bezeichners kann ein Buchstabe oder das Unterstreichzeichen sein.

Formale Syntax:

```
Bezeichner  =   Buchstabe { Buchstabe | Ziffer }
Buchstabe   =   'a' | 'b' |  ...  | 'z' | 'ä' | 'ö' | 'ü' | '_'
Ziffer      =   '0' | '1' | '2' |  ...  | '9'
```

Beachten Sie bitte, daß es sich hierbei um eine lexikalische Definition handelt (siehe Abschnitt 1.1) und daß auch hier nicht zwischen Groß- und Kleinbuchstaben unterschieden wird. Was in der Syntax nicht zum Ausdruck kommt: Wortsymbole wie z. B. "BEGIN" sind keine Bezeichner.

2.2 Programmkopf

Die Zeile "Program ..." kann entfallen. Wenn sie aber gesetzt wird, ist die Syntax folgende:

```
"PROGRAM" Bezeichner [ "(" Bezeichner { "," Bezeichner } ")" ] [ ";"s ]
```

Im Klartext: Nach dem Schlüsselwort "Program" muß stets ein Bezeichner folgen. Optional kann man dann eine in runden Klammern eingeschlossene Parameterliste setzen, bestehend aus durch Komma getrennten Bezeichnern. Achtung: der Inhalt dieser Liste wird völlig ignoriert und hat keinerlei Bedeutung! Wie fast immer, ist auch hier das ";" am Schluß optional.

2.3 Blöcke

Ein Block bildet in PASCAL den Rumpf eines Programms bzw. einer Procedure oder Function und besteht aus:

- Label-Deklarationsteil
- Konstanten-Definitionsteil
- Typdefinitionsteil
- Variablendeklarationsteil
- Procedure- und Function-Deklarationen
- Bibliotheksdefinitionen
- Anweisungsteil

Der letzte Teil ist der einzige, der immer vorhanden sein muß, alle anderen sind optional.

Bei KICK-PASCAL ist die Reihenfolge der Deklarations- und Definitionsteile beliebig; jeder Teil kann auch beliebig oft vorkommen. Wichtig ist nur, daß am Ende der mit BEGIN ... END umschlossene Anweisungsteil kommt.

Folgendes wäre ein korrektes KICK-PASCAL-Programm:

```
{ "Program"-Kopf weggelassen }

Const diff=7;

Procedure p(Var i:integer);
  Begin
     writeln(i); i:=i-diff
  End;

Type ttyp=integer;

Var v: ttyp;

Const con=0815;

Begin
   v:=con; p(v); p(v)
End.
```

Der Punkt am Programmende ist übrigens auch nicht unbedingt nötig.
Formal sieht das Ganze dann so aus:

```
Deklarationsteil  =   { Labeldeklaration  |  Konstantendefinition |
                      Typdefinition  | Variablendeklaration |
                      Prozedurdeklaration  | Librarydefinition

Block             =   Deklarationsteil  Anweisungsteil

Programm          =   [ Programmkopf ] Uses-Deklaration Block
                      [ "." ]
```

Zu den meisten hier auftretenden Begriffen wie "Variablendeklaration", "Library-definition" oder "Anweisungsteil" finden Sie im weiteren Verlauf dieses Handbuchs ausführliche Erläuterungen. Was eine "Uses-Deklaration" ist, erfahren Sie im Kapitel über Units.

2.4 Label

Jedes Label, das benutzt wird, muß zuvor deklariert werden. Die Syntax eines Label-Deklarationsteils ist:

```
Labeldeklaration  =   "LABEL" Marke { "," Marke }  [ ";" ]

Marke             =   Bezeichner | Ziffernfolge
```

Ein Label ("Marke") ist entweder eine Integer-Konstante oder ein Bezeichner. Das Label muß auf der Ebene deklariert werden, auf der es gesetzt wird: Man darf z. B. nicht ein Label im Hauptprogramm deklarieren und dann im Anweisungsteil einer Prozedur setzen.

2.5 Konstanten und Konstantendefinition

Syntax für Konstantendefinition:

```
"CONST" Name "=" Konstante { { ";" } Name "=" Konstante } [ ";" ]
```

"Name" ist ein Bezeichner, "Konstante" normalerweise ein Literal. Näheres entnehmen Sie bitte den Kapiteln über die jeweiligen Datentypen.

Eine Konstante kann in Kickpascal aber auch ein Ausdruck sein, in dem nur konstante Operanden und bestimmte Operatoren auftreten. Im Ganzzahl-Bereich können alle einstelligen (NOT, -) und zweistelligen (+, *, mod, shr, ...) Operatoren in Konstanten benutzt werden, außerdem die folgenden Funktionen (natürlich nur mit konstanten Parametern):

```
Abs   Chr   Ord   Pred   SizeOf   Sqr   Succ
```

Succ, Pred und Ord werden auch bei konstanten Parametern der anderen geordneten Datentypen (Char, Boolean, Aufzählungstypen) ausgewertet.

Real- und String-Ausdrücke werden nicht vom Compiler ausgewertet und können deshalb nicht als Konstante benutzt werden. Die einzigen Operatoren, die bei Real-Konstanten erlaubt sind, sind „+" und „-" als Vorzeichen. Also sind

```
Const n = 17 + 4;
      nQuadrat = Sqr (n);
      R = 17.4;
      minusR = -R;
      s = '17 plus 4';
```

korrekt definierte Konstanten,

```
Const r = 17 + 0.4;              { Real-Ausdruck! }
      s = ' Guide' + '42';       { String-Ausdruck! }
```

jedoch nicht.

Außer in Konstantendefinitionen treten Konstanten noch an einigen anderen Stellen auf, z. B. in Typdeklarationen (Ausschnittstypen, z. B. als ARRAY-Indextyp) und als Sprungmarke in der CASE-Anweisung.
Auch hier gelten die selben Regeln für Konstanten-Ausdrücke, so daß also Deklarationen wie

```
VAR Feld: Array[1..40+2] of Set of 0 .. 1 shl 6 - 1;
```

erlaubt sind.

2.6 Typdefinition

In PASCAL kann man sich Datentypen selbst definieren. Um nicht immer wieder dieselbe Typbeschreibung schreiben zu müssen, kann man dem Datentypen einen Namen geben. Dies geschieht im Typ-Definitionsteil eines Blocks: Auf der linken Seite eines Gleichheitszeichens steht der gewünschte Name, auf der rechten Seite die Typbeschreibung. Von dieser Stelle an ist der definierte Typ unter diesem Namen im aktuellen Block verfügbar.

Syntax für Typdefinitionsteil:

```
"TYPE" Name "=" Typ { [ ";" ] Name "=" Typ }  [ ";" ]
```

Den zahlreichen Datentypen ist ein Extra-Kapitel dieses Handbuchs gewidmet.

2.7 Variablendeklaration

Jede Variable, die benutzt wird, muß zuvor deklariert werden. Dies geschieht im Variablen-Deklarationsteil eines Blocks. Jede Variable hat einen Namen, der (natürlich) ein Bezeichner ist, und einen Datentypen, z. B. "Integer", "Real" oder auch ein selbstdefinierter Datentyp.

Zur Syntax:

```
Variablendeklarationsteil      =   "VAR" { Variablendeklaration
                                     [ ";" ] }

Variablendeklaration           =
     Namensliste ":" Typ  [ [ ";" ] ("IMPORT"|"EXPORT"|"STATIC") ]
     Name ":" Typ "ABSOLUTE" Konstante

Namensliste                    =   Name { "," Name }

Name                           =   Bezeichner
```

Es gibt also mehrere Arten von Variablen:

- Die normalen Standardpascal-Variablen. Sie werden durch eine mit ":" abgeschlossene Bezeichnerliste und dahinterstehendem Typ deklariert.

  Beispiel: `a, b, abc: ARRAY [26..731] OF Char;`

  Der Compiler wird angewiesen, zur Laufzeit beim Betreten des Blocks, in dem diese Variablen deklariert sind, auf dem Stack für die Variablen Platz einzurichten.

- Pseudo-Variablen, die an feste Speicheradressen gebunden sind. Hier besteht die Deklaration aus einem einzelnen (!) Bezeichner, dem wie üblich ein Doppelpunkt und ein Datentyp folgen. Dahinter stehen dann (ohne trennendes Semikolon!) die Directive "Absolute" und die gewünschte Speicheradresse in Form einer ganzzahligen Konstanten.

  Beispiel: `Hintergrund: integer ABSOLUTE $dff180;`

  Solche Variablen sind vor allem bei der Programmierung der Hardware interessant: Im obigen Beispiel entspricht die Variable "Hintergrund" dem Speicherwort an der Adresse $dff180 und somit dem Register des Grafikchips, das die Hintergrundfarbe des Bildschirms enthält. Wenn Sie dieser Variablen einen Wert zuweisen, ändert sich die Farbe entsprechend - einmal davon abgesehen, daß das Betriebssystem Ihnen dazwischenpfuscht und in schneller Folge den alten Wert wiederherstellt.

Es gibt zwei vordefinierte ABSOLUTE-Variablen: "Mem" ist definiert als "Mem: Array [0..MaxLongInt] of Byte Absolute 0", repräsentiert also den gesamten Hauptspeicher, und "SysBase" enthält durch ihre Deklaration als "SysBase: Ptr Absolute 4" einen Zeiger auf die ExecBase-Struktur.

Solche maschinennahen Programmierungen dürften die einzigen sinnvollen Anwendungen von ABSOLUTE-Variablen sein, denn ein anständiger Programmierer benutzt niemals feste Adressen - schon gar nicht auf dem AMIGA!

- Statische Variablen, die im Gegensatz zu den normalen Variablen bereits beim Programmstart eingerichtet werden. Sie belegen keinen Stack-Platz, sondern werden an anderer geeigneter Stelle im Speicher abgelegt. Das macht sich bemerkbar, wenn eine statische Variable lokal in einer rekursiven Prozedur oder Funktion deklariert werden. Bei einer Rekursion wird für sie dann keine "neue" Variable eingerichtet.

  Variablen werden als statisch deklariert, indem man hinter ihre Deklarationszeile die Directive "STATIC" setzt.

  Beispiel:
  ```
  VAR y, x: Real; STATIC;
      s: STRING[20]; STATIC;
  ```

  Im Kapitel "Units und Module" ist den statischen Variablen ein Abschnitt gewidmet.

- Modulüberschreitende importierte oder exportierte Variablen. Diese Variablen sind immer automatisch statisch. Näheres über diese Variablenart finden Sie im Kapitel über Module.

2.8 Prozeduren und Funktionen

PASCAL kennt zwei Arten von Unterprogrammen: Procedures und Functions. Sie dienen jeweils dazu, ein Programm in kleinere Einheiten zu gliedern und so überschaubarer zu machen. Außerdem spart man sich Tipparbeit, wenn man einen öfter identisch oder ähnlich auftretenden Programmteil als Prozedur deklariert. Es gibt aber auch viele andere Probleme, z. B. inhärent rekursive, die man ohne Prozeduren und Funktionen kaum lösen könnte.

2.8.1 Allgemeines über Prozeduren

Eine Prozedur wird im Deklarationsteil eines Blocks deklariert. Sie wird mit dem Prozedurkopf eingeleitet, der im einfachsten Fall aus dem Wortsymbol "PROCEDURE" und dem Namen besteht und mit einem Semikolon abgeschlossen werden sollte:

```
PROCEDURE Beispiel;
```

Danach folgt der Prozedurrumpf, ein Block. Eine Prozedur kann lokale Variablen, Konstanten, Datentypen usw. haben. Es ist sogar möglich, Prozeduren und Funktionen beliebig tief ineinander zu verschachteln (genaugenommen ist diese Schachtelungstiefe in KICK-PASCAL auf die Tiefe 16 beschränkt, aber in der Praxis wird man wohl kaum über eine Tiefe von 3 oder 4 hinauskommen).

Eine Prozedur wird aufgerufen, indem man an der aufrufenden Stelle einfach ihren Namen setzt.

Ein Beispiel:

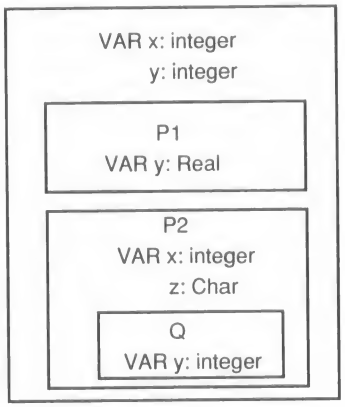
```
PROGRAM ProzedurDemo;

  VAR x, y: integer;

  PROCEDURE P1;
     VAR y: Real;
     BEGIN
       y:= 42;
       writeln ('Prozedur P1', x:10, y:10);
     END;

  PROCEDURE P2;
     VAR x: integer;
         z: Char;

    PROCEDURE Q;
       { lokale Prozedur von "P2" }
       VAR y: integer;
       BEGIN
         x:= 1;
         y:= 2;
         z:= '?';
         writeln('Prozedur Q', x:10, y:10, z:10)
       END;   { Ende von "Q" }

   BEGIN  { Anweisungsteil von Prozedur P2 }
      x:= 26;
      y:= 731;
      z:= 'A';
      writeln('Anfang P2', x:10, y:10, z:10);
      Q;
      writeln('P2 nach Q', x:10, y:10, z:10);
      P1;
      writeln('Ende P2  ', x:10, y:10, z:10)
    END;

  BEGIN  { Hauptprogramm }
     x:= 47;
     y:= 11;
     writeln('Programmstart       ', x:10, y:10);
     P1;
     writeln('Mitte Hauptprogramm', x:10, y:10);
     P2;
     writeln('Programmende        ', x:10, y:10)
   END.
```

Dieses Programm gibt folgendes aus:

Programmstart	47	11	
Prozedur P1	47	42	
Mitte Hauptprogramm	47	11	
Anfang P2	26	731	A
Prozedur Q	1	2	?
P2 nach Q	1	731	?
Prozedur P1	47	42	
Ende P2	1	731	?
Programmende	47	731	

Wie funktioniert das Ganze? Wir haben ein Hauptprogramm mit zwei Prozeduren, von denen eine wiederrum eine lokale Prozedur enthält. Alle Prozeduren haben lokale Variablen. Man kann sich das so vorstellen:

Beispiel



Zunächst einmal fällt auf, daß Variablen scheinbar doppelt deklariert werden und das auch noch mit verschiedenen Datentypen! Das hängt mit den folgenden Regeln über die Gültigkeitsbereiche von Bezeichnern zusammen: (Wichtig! Aufmerksam durchlesen, tief Luft holen und darüber nachdenken!)

1. Das Hauptprogramm und alle Prozeduren haben jeweils eine eigene "Schachtel" für ihre Bezeichner. Da Prozeduren wieder selbst als lokale Prozeduren verschachtelt werden können, gibt es eine Hierarchie von Schachteln: Ganz "oben" ist die Schachtel des Hauptprogramms, unter der die Schachteln, der im Hauptprogramm deklarierten Prozeduren stehen, unter welchen jeweils die darin lokal deklarierten Schachteln stehen usw.

2. In jeder Schachtel sind die Bezeichner aus höheren Schachteln bekannt. Sie können aber innerhalb dieser Schachtel "überdefiniert" werden. Eine obere Schachtel merkt grundsätzlich nichts von den Bezeichnern der ihr untergeordneten Schachteln.

3. Ein Bezeichner ist "lokal" zu der Schachtel, in der er deklariert wird und "global" zu allen tiefer liegenden Schachteln. In höheren Schachteln ist er nicht bekannt.

Nun können wir anfangen, das Beispielprogramm Schritt für Schritt durchzugehen. Zunächst wäre da das Hauptprogramm mit den Variablen "x" und "y", jeweils vom Typ "Integer". Von den Variablen, die in den Prozeduren deklariert werden, "weiß" das Hauptprogramm nichts, so daß die Variablen "x" und "y" zunächst ganz normal einen Wert erhalten und ausgegeben werden.

Jetzt wird aber die Prozedur "P1" aufgerufen. Dort ist eine Variable "y" definiert, und es wird die interessante Wertzuweisung "y:= 42" ausgeführt. Die lokale Definition hat Vorrang vor der globalen, also bezieht sich diese Wertzuweisung auf die innerhalb "P1" deklarierte Real-Variable. Diese Variable wird dann auch ausgegeben, ebenso eine Variable "x". Dazu gibt es im Moment nur eine Deklaration: die globale Integer-Variable des Hauptprogramms. Also wird deren Wert (47) ausgegeben.

Nach Ende des Anweisungsteils von "P1" kehren wir in das Hauptprogramm zurück. Die Variablen "x" und "y" haben noch dieselben Werte wie zuvor (47 und 11). In "P1" wurde zwar eine Wertzuweisung an "y" vorgenommen, diese bezieht sich aber auf die dortige lokale Variable, die mit dem gleichnamigen globalen "y" nichts zu tun hat.

Nun wird die Prozedur "P2" ausgeführt. Dort werden als erstes Wertzuweisungen an "x", "y" und "z" durchgeführt. Analog zum ersten Prozeduraufruf können Sie sich klarmachen, daß "x" und "z" lokale Variabeln sind, während "y" sich hier auf den globalen Bezeichner bezieht. Es wird hier also in einer Prozedur der Wert einer globalen Variablen geändert. So etwas nennt man einen "Seiteneffekt" der Prozedur (also eine auf den ersten Blick nicht erkennbare Auswirkung der Prozedur auf den Gesamt-Programmablauf) und sollte möglichst vermieden werden - im nächsten Abschnitt erfahren Sie, wie man so etwas mit Hilfe von "Parametern" sauber machen kann. Jedenfalls wird hier 26, 731 und "A" ausgegeben, bevor "Q" ausgeführt wird. Dort passiert nicht mehr viel Neues: Eine lokale Variable "y" wird eingerichtet, und es gibt einen Seiteneffekt auf die Variablen "x" und "z" der Prozedur "P2". Nach Rückkehr aus "Q" nach "P2" haben diese Variablen also plötzlich einen anderen Wert.

Nun geschieht etwas Interessantes: Die Prozedur "P1" wird aus "P2" heraus aufgerufen. Das ist möglich, denn der Bezeichner "P1" ist ja global, d. h. in der Schachtel des Hauptprogramms, deklariert und somit in der Schachtel von "P2" bekannt. Dort wird wieder die wohlbekannte Zuweisung an das lokale "y" durchgeführt und "x" ausgegeben. Welches "x" ist nun gemeint - das des Hauptprogramms oder das der aufrufenden Prozedur "P2"? In PASCAL gibt es den Begriff des "statischen" und des "dynamischen" Vorgängers einer Prozedur. Statischer Vorgänger von "P1" ist immer das Hauptprogramm, denn darin ist die Prozedur ja deklariert. Dynamischer Vorgänger ist jeweils die aufrufende Schachtel. In diesem Fall, also beim zweiten Aufruf, ist die Prozedur "P2" dynamischer Vorgänger. Die Identifikation

von Bezeichnern wird bereits vom Compiler vorgenommen, muß sich also auf den statischen Vorgänger beziehen, denn der dynamische steht ja erst zur Laufzeit der Prozedur fest. Also bezieht "x" sich hier wieder auf die Variable des Hauptprogramms und nicht die des dynamischen Vorgängers "P2", so daß wieder 47 ausgegeben wird.

Da "P1" keine Seiteneffekte hat, haben sich nach Rückkehr von dort in die Schachtel von "P2" keine Variablenwerte geändert. Nach dem Ende von "P2" und dem Rücksprung ins Hauptprogramm hat es aber einen Seiteneffekt auf die globale Variable "y" gegeben, so daß jetzt ein anderer Wert ausgegeben wird.

In der Praxis wird man natürlich keine so komplizierten Schachtelungen verwenden und nur in Ausnahmefällen globale Bezeichner überdefinieren.

2.8.2 Parameter

Es wurde bereits erwähnt, daß man Seiteneffekte aus Gründen der Lesbarkeit des Listings vermeiden sollte. Um in der Deklaration der Prozedur deutlich zu machen, wovon der Prozedurverlauf abhängt und welche globalen Größen er beeinflußt, gibt es die Parameter.

PASCAL kennt zwei Arten von Parametern: Wertparameter ("Call by value") und Variablenparameter ("Call by reference"). Andere exotische Programmiersprachen kennen auch "Call by name" (z. B. Simula, Macros in Assembler) oder "Call by value and return" (z. B. Fortran), aber es ist gewiß kein Verlust, daß uns jene Übergabearten in PASCAL erspart bleiben.

Der Witz hat zwar einen Bart, aber hier ist er trotzdem für alle, die ihn noch nicht kennen: PASCAL-Erfinder Niklaus Wirth auf die Frage eines Amerikaners, wie man denn seinen Namen aussprechen solle: "You can call me by name, Wirth, or by value, Worth."

2.8.2.1 Wert-Parameter

Das folgende Programm enthält eine Prozedur, die den größten gemeinsamen Teiler ("ggT") zweier Zahlen nach dem Euklidischen Algorithmus berechnet und ausgibt. Damit die Prozedur "weiß", welche beiden Zahlen gemeint sind, werden diese als Parameter übergeben:

```
PROGRAM Euklid;

   PROCEDURE ggT (a, b: integer);
     VAR hilf: integer;
      BEGIN
        WHILE a<>0 DO
          BEGIN
            hilf:= abs (a);
            a:= b MOD a;
            b:= hilf;
          END;
        writeln(b)
      END;
```

```
BEGIN
   write ('Der ggT von 17 und 4 ist ');  ggT (17, 4);
   write ('Der ggT von 12 und 16 ist ');  ggT (12, 16);
   write ('Der ggT von 42 und 4711 ist ');  ggT (42, 4711);
END.
```

Man kann also in dem Prozedurkopf eine sog. Parameterliste aufnehmen. In unserem Beispiel werden in der Prozedur zwei lokale Variablen "a" und "b" des Typs Integer eingerichtet. Beim Aufruf der Prozedur ist dann anzugeben, mit welchen Werten diese Variablen vorbesetzt werden sollen: Beim ersten Aufruf erhält "a" den Wert 17 und "b" wird 4 zugewiesen usw. Diese Parameter verhalten sich ansonsten wie ganz normale lokale Variablen der Prozedur, z. B. kann man ihnen - wie im Beispiel geschehen - einen neuen Wert zuweisen usw.

Die "aktuellen Parameter" (also die Werte, die man beim Prozeduraufruf angibt) können beliebig komplexe Ausdrücke sein, sie müssen aber zu den "formalen Parametern" (also dem, was in der Prozedurdeklaration in der Parameterliste steht) wertzuweisungskompatibel sein.

Wenn in der formalen Parameterliste mehrere unterschiedliche Parameter stehen sollen, sind diese mit einem Semikolon zu trennen, z. B. so:

```
PROCEDURE MultiPara (Zahl: Real; c1, c2: Char; Flag: Boolean);
```

Beim Aufruf müssen die aktuellen Parameter aber nichtsdestotrotz durch Komma getrennt werden, etwa:

```
MultiPara (x+2, 'A', chr(i+32), b);
```

In der formalen Parameterliste müssen Datentypen grundsätzlich in Form eines Typbezeichners angegeben werden:

```
PROCEDURE Pr(St: STRING[50]);
```

wäre also FALSCH! "STRING" allein ohne Längenangabe (d. h. mit Default 80) ist streng genommen kein Typbezeichner. Es kann hier aber als solcher verwendet werden.

2.8.2.2 Variablenparameter

Angenommen, wir wollen den ggT nicht ausgeben, sondern weiter mit ihm rechnen. Beispielsweise könnten wir ihn brauchen, um einen Bruch zu kürzen. Es gibt verschiedene Arten, wie eine Prozedur einen Wert zurückgeben kann: Seiteneffekte auf globale Variablen (sollte man besser sein lassen), Funktionen (siehe nächster Abschnitt) und VAR-Parameter.

In unserem ggT-Beispiel könnte das so aussehen:

```
PROCEDURE ggT (a, b: integer; VAR Ergebnis: integer);
   VAR hilf: integer;
   BEGIN
     WHILE a<>0 DO
       BEGIN
         hilf:= abs (a);
         a:= b MOD a;
         b:= hilf;
       END;
     Ergebnis:= b;
   END;
```

Der Variablen-Parameter wird also wie ein Wert-Parameter deklariert, nur daß davor das Schlüsselwort "VAR" gesetzt wird. In der aktuellen Parameterliste ist für den VAR-Parameter eine dazu typkompatible Variable anzugeben, etwa so:

```
VAR n: integer;

...

ggT(42, 4711, n);
writeln('Der ggT von 42 und 4711 ist ', n);
```

Beim Aufruf der Prozedur "ggT" erhält der formale Parameter "Ergebnis" eine "Referenz" auf die aktuelle Parameter-Variable "n". Das bedeutet: eine Variable namens "Ergebnis" gibt es überhaupt nicht, sie ist nur ein anderer Name für den formalen Parameter. Wenn an "Ergebnis" ein Wert zugewiesen wird, erhält in Wirklichkeit der aktuelle Parameter einen neuen Wert. Die beiden Variablen sind für die Dauer der Prozedurabarbeitung identisch.

Ein anderes Beispiel: Eine Prozedur soll einen Bruch, der als Nenner und Zähler übergeben wird, kürzen. Wir verwenden dazu wieder unsere "ggT"-Prozedur.

```
PROGRAM Bruch;

  VAR nenn, zaeh: integer;

  PROCEDURE ggT (a, b: integer; VAR Ergebnis: integer);
     VAR hilf: integer;
     BEGIN
       WHILE a<>0 DO
         BEGIN
           hilf:= abs (a);
           a:= b MOD a;
           b:= hilf;
         END;
       Ergebnis:= b;
     END;

  PROCEDURE Kürzen (VAR zähler, nenner: integer);
     { Die Parameter durch ihren ggT teilen. Wenn nenner=0,
       Fehler melden. Bei zähler=0 den Nenner auf 1 setzen. }
     VAR g: integer;
     BEGIN
       IF nenner=0 THEN
         error ('Zähler ist Null!')
```

```
      ELSE
        IF zähler=0 THEN
          nenner:= 1
        ELSE
          BEGIN
            ggT (zähler, nenner, g);
            nenner:= nenner DIV g;
            zähler:= zähler DIV g
          END
    END;

BEGIN  { Hauptprogramm}
   write('Bruch eingeben: '); readln (zaeh, nenn);
   Kürzen (zaeh, nenn);
   writeln('gekürzt: ', zaeh, ' / ', nenn)
END.
```

Das Programm tut das Gewünschte.

Wir haben es wirklich mit einem "Call by reference" zu tun, nicht mit "Call by name". Beispiel:

```
PROGRAM Kleinlich;
   VAR i: integer;
       a: ARRAY [ 1..2 ] OF integer;

  PROCEDURE Proc (VAR Para: integer);
     BEGIN
       i:= 2;
       Para:= 42
     END;

  BEGIN
     a[1]:= 1; a[2]:= 2;
     i:= 1;
     Proc (a[i]);
     writeln(a[1]: 5, a[2]: 5)
     END.
```

Quizfrage: Was gibt dieses Programm aus?
Antwort: `42`

Beim Aufruf erhält "Para" eine Referenz auf "a[i]" - und das ist zu diesem Zeitpunkt "a[1]". Deshalb wird "42" auch an "a[1]" zugewiesen, obwohl inzwischen "i" einen anderen Wert erhalten hat. Beim sog."Call by name" würde einfach für "Para" jeweils "a[i]" eingesetzt, so daß dann "a[2]" den Wert 42 erhalten würde.

2.8.3 Funktionen

Funktionen sind Prozeduren, die einen Wert zurückgeben. Der Aufruf einer Funktion ist daher kein Befehl, sondern Teil eines Ausdrucks.

Ihre Deklaration ist identisch mit der Prozedur-Deklaration, aber statt "PROCEDURE" ist das Schlüsselwort "FUNCTION" zu verwenden und am Ende des Kopfs ist der Typ des Rückgabewerts anzugeben:

```
FUNCTION ggT (a, b: integer): integer;
   VAR hilf: integer;
   BEGIN
     WHILE a<>0 DO
       BEGIN
         hilf:= abs (a);
         a:= b MOD a;
         b:= hilf;
       END;
     ggT:= b;
   END;
```

Irgendwo im Anweisungsteil der Funktion - oder einer untergeordneten lokalen Prozedur oder Funktion - muß der Rückgabewert definiert werden, indem er dem Funktionsnamen zugewiesen wird.

Ein Aufruf dieser Function könnte so aussehen:

```
writeln('Der ggT von 42 und 4711 ist: ', ggT(42,4711));
```

Oder so:

```
nenner:= nenner DIV ggT(nenner,zähler);
```

Der Rückgabetyp muß in Form eines Typbezeichners angegeben werden. Auch hier ist "STRING" als Pseudo-Bezeichner möglich.

Während Parameter grundsätzlich von jedem beliebigen Typ sein können, sind die Rückgabewerte von Functions auf "skalare" Typen beschränkt. Erlaubt sind:

- Numerische Datentypen (ganzzahlig und Gleitpunkt)
- CHAR, STR, Stringtypen
- BOOLEAN
- Aufzählungs- und Ausschnittstypen
- Pointertypen

2.8.4 Prozedurparameter und andere Spezialitäten

KICK-PASCAL kennt keine Conformant Array Parameter. Dieses empfohlene Extra des ISO-Standards ermöglicht es, Arrays mit variablem Indexbereich als Parameter zu übergeben. Man kann sich aber mit typfreien Zeigern behelfen (siehe Kapitel "Pointer").

Beispiel:

```
PROGRAM CAP_Ersatz;
 VAR i: integer;
     a: ARRAY[1..10] OF integer;

  PROCEDURE Sort(Arrayzeiger: Ptr; FeldAnzahl: integer);
     VAR ap: ^ARRAY[0..MaxInt] of integer;
         i, j: integer;
     BEGIN
       ap:= Arrayzeiger;
```

```
        { Achtung, Array geht von 0 bis Feldanzahl-1 }
        FOR i:= FeldAnzahl-2 DOWNTO 0 DO
          FOR j:= 0 TO i DO
            IF ap^[ j ] > ap^[ j+1 ] THEN
              Exchange(ap^[ j ], ap^[ j+1 ])
      END;

    BEGIN
      FOR i:=1 TO 10 DO a[i]:= random(1000);
      Sort(^a, 10);
      FOR i:=1 TO 10 DO writeln(a[ i ]);
    END.
```

Dieses Programm simuliert einen Conformant Array Parameter, indem ein Zeiger auf das Array (KICK-PASCAL-Spezialität!) und die Anzahl der Elemente übergeben wird. Beachten Sie aber, daß sich der Indexbereich ändert: a[1] entspricht ap^[0].

Man kann auch Prozeduren und Funktionen als Parameter übergeben. Formal sieht das so aus, daß man in die formale Parameterliste einen Prozedur- oder Funktionskopf schreibt. In der aktuellen Parameterliste ist dann nur noch der Name der Prozedur oder Funktion anzugeben.

Beispiel:

```
PROGRAM FuncParam;

  PROCEDURE ab(FUNCTION f(x:Real):Real);
    VAR i:integer;
    BEGIN
      FOR i:=0 TO 10 DO
        writeln(i/5:5:1, f(i/5):12:6);
      writeln;
    END;

  BEGIN
    writeln('Sinus:');   Tab(Sin);
    writeln('Cosinus:'); Tab(Cos)
  END.
```

Dieses Programm druckt Wertetabellen der Funktionen Sin und Cos. Ein weiteres Beispiel zu diesem Thema finden Sie auf der Diskette unter dem Namen "integra.p".

Bei rekursiven Problemen kommt es manchmal vor, daß zwei Prozeduren oder Funktionen sich gegenseitig aufrufen müssen. Die einfachste Lösung wäre, wenn man die Prozeduren ineinander verschachtelt. Falls das aus irgendwelchen Gründen nicht möglich ist, kann man das Problem über "FORWARD"-Referenzen lösen.

Dabei schreibt man von der ersten Prozedur oder Funktion zunächst nur den Kopf, gefolgt von "FORWARD". Nun "weiß" der Compiler, daß es diese Prozedur gibt und wie ihre Parameterliste aussehen soll. Nun kann man die zweite Prozedur und was auch immer schreiben, bevor man irgendwann vor Ende des Deklarartionsteils den Block der ersten Prozedur "nachträgt". Er wird eingeleitet mit dem Kopf dieser Prozedur, aber ohne Parameterliste und (bei Funktionen) Ergebnistyp.

Das folgende Beispiel enthält zwei Funktionen, die auf besonders umständliche Weise feststellen, ob eine Zahl gerade bzw. ungerade ist:

```
PROGRAM Forwarded;

VAR i: integer;

FUNCTION Ung(n: integer): Boolean; FORWARD;

FUNCTION Ger(n: integer): Boolean;
   BEGIN
     { "n" ist gerade, wenn n=0 gilt oder n-1 ungerade ist. }
     IF n=0 THEN
       Ger:= true
     ELSE
       Ger:= Ung(n-1)
   END;

FUNCTION Ung;
   BEGIN      { Eine Zahl ist ungerade, wenn ihr Betrag nicht
                gerade ist. }
     Ung:= NOT Ger(abs(n))
   END;

BEGIN
   write ('Bitte ganze Zahl eingeben: ');
   readln (i);
   IF ger(i) THEN
     writeln('Gerade.')
   ELSE
     writeln('Ungerade.')
END.
```

2.8.5 Zusammenfassend: Syntax

Hier ist die vollständige Syntax für Prozedur- und Funktions-Deklarationen:

```
Prozedur/Funktionsdeklaration
                  =     ProzFunkKopf (("FORWARD" | "IMPORT" |
                        "EXTERNAL") ";" | ["EXPORT" ";"]  Block)

ProzFunkKopf      =     ("PROCEDURE" Bezeichner [ Formale
                        Parameterliste ] [ ";" ])  | ("FUNCTION"
                        Bezeichner [ FormaleParameterliste ]
                        [ ":"  Typbezeichner ] [ ";"  ])

FormaleParameterliste
                  =     "(" FormalerParameter { ";" Formaler
                        Parameter } ")"

FormalerParameter =     ([ "VAR" ]  Bezeichner { "," Bezeichner } ":"
                        TypBezeichner)  | ProzFunkKopf
```

Zu "IMPORT", "EXPORT" und "EXTERNAL" finden Sie im Linker-Kapitel nähere Erläuterungen.

2.9 Anweisungsteil

Der Anweisungsteil eines Blocks besteht aus einer mit BEGIN...END geklammerten Folge von Befehlen. Er ist der einzige Teil, der in einem Block nie fehlen darf. Syntax:

```
Anweisungsteil =  "BEGIN"  Anweisung { [ ";" ] Anweisung }  "END"
```

Es gibt viele verschiedene Arten von Anweisungen, was auch in unserer EBNF-Syntax zum Ausdruck kommt:

```
Anweisung = [ Marke ":" ] [ Wertzuweisung | Prozeduraufruf |
            Verbundanweisung | Fallunterscheidung | REPEAT-Schleife |
            WHILE-Schleife | FOR-Schleife | CASE-Anweisung |
            WITH-Anweisung | GOTO-Sprung  ]
```

Vor jeder Anweisung kann also eine Sprungmarke stehen, und eine Anweisung kann grundsätzlich auch leer sein.

2.9.1 Wertzuweisung

Syntax: `Variable ":=" Ausdruck`

Der Ausdruck muß zur Variablen wertzuweisungskompatibel sein. Dabei gelten folgende Regeln:

Variable	Ausdruck
Ganzzahlig	beliebiger Ganzzahltyp, auch andere "Breite"
Real, Double	numerisch
Char	Char-Konstante, -Variable, -Funktion
Boolean	Boolean
Aufzählungstyp	Ausdruck desselben Aufzählungstypen
Pointer	typfreier Zeiger, "NIL" oder Zeiger mit typkompatiblem Referenztyp
Str	Char- oder Str-Ausdruck, STRING-Variable, String-Konstante, keine String-Ausdrücke!
SET	beliebiger Mengenausdruck
STRING[n]	Char- oder Stringausdruck
ARRAY	Array-Variable mit gleichem Indextyp und typkompatiblem Elementtyp
RECORD	gleicher Recordtyp, oder strukturgleicher Record mit jeweils typkompatiblen Feldern

In der Tabelle taucht des öfteren der Begriff der "Typkompatibilität" auf. Diese Kompatibilität ist strenger definiert als der Begriff der Wertzuweisungskompatibilität: hier müssen skalare Typen zusätzlich die gleiche "Breite" haben. Falls Sie in der Übersicht die Ausschnittstypen vermißt haben: Sie werden mit ihren jeweiligen Grundtypen identifiziert. Ein Datentyp wie "1..10000" wäre also sowohl wertzuweisungs- als auch typkompatibel zu "Integer".

2.9.2 Prozeduraufruf

Es gibt vordefinierte Standardprozeduren (siehe Kapitel V) und selbstdefinierte Prozeduren, wie im Abschnitt 8 dieses Kapitels dargestellt wurde. Beim Aufruf sind diese beiden Prozedurtypen völlig gleichgestellt:

```
Prozeduraufruf = ProzedurBezeichner [ Parameterliste |
                                      Write-Parameterliste e ]

Parameterliste = "(" Ausdruck { "," Ausdruck } ")"
```

Abgesehen von "Write" und "WriteLn" sehen Parameterlisten also stets gleich aus. Allerdings ist es bei vordefinierten Prozeduren oft so, daß der genaue Inhalt von Parameterlisten, z. B. Anzahl oder Datentyp der Parameter, nicht fest ist.
Alles über vordefinierte Prozeduren finden Sie im Kapitel V.

2.9.3 Verbundanweisug

Man kann mehrere Anweisungen zu einer einzigen zusammenfassen, indem man sie mit "BEGIN" ... "END" klammert:

```
Verbundanweisung =  "BEGIN" Anweisung { | ";" ] Anweisung } "END"
```

Vor dem "END" muß also kein Semikolon stehen. Man darf es natürlich setzen, was insofern in der Syntax enthalten ist, da man sich als letzte Anweisung eine Leeranweisung denkt.

Verbundanweisungen werden bei den im folgenden beschriebenen strukturierten Anweisungen benötigt, da an einigen Stellen formal keine Anweisungsfolge, sondern eine einzelne Anweisung stehen muß.

2.9.4 Fallunterscheidung

Die wichtigste Art der Fallunterscheidung ist die "IF"-Anweisung. Ihre Syntax ist wie folgt:

```
"IF" Ausdruck "THEN" Anweisung [ "ELSE" Anweisung ]
```

Als Bedingung ist ein beliebiger Ausdruck des Typs "Boolean" erlaubt. Insbesondere kann hier eine einzelne Variable dieses Typs gesetzt werden. Ich weise darauf extra hin, da es ein "beliebter" Stil-Fehler ist,

```
IF b = true THEN ...
```

zu schreiben (wobei "b" eine boolesche Variable ist). Hier reicht ganz einfach:

```
IF b THEN ...
```

Ist der Ausdruck "wahr", wird die Anweisung hinter dem "THEN" ausgeführt (mit Hilfe einen Verbundanweisung kann man hier natürlich auch eine beliebig lange Anweisungsfolge setzen). Ist die Bedingung nicht wahr, so wird - falls vorhanden - die Anweisung des "ELSE"-Teils ausgeführt.

Es gibt eine kleine "Zweideutigkeit" in diesem Zusammenhang:

```
IF b1 THEN
   IF b2 THEN s1
   ELSE s2;
```

Diese Zeile ist - syntaktisch betrachtet - nicht eindeutig: Das "ELSE" könnte theoretisch zu beiden "IF" gehören. Man hat hier definiert, daß es zum letzten "IF" gehört. Die obige Anweisung entspricht also

```
IF b1 THEN
   BEGIN
     IF b2 THEN s1
           ELSE s2
   END;
```

Um das "ELSE" ausdrücklich an das erste "IF" zu binden, könnten Sie folgendes schreiben:

```
IF b1 THEN
   BEGIN
     IF b2 THEN s1
   END
ELSE
  s2;
```

2.9.5 REPEAT-Schleife

Syntax:

```
"REPEAT" Anweisung { ";" Anweisung } "UNTIL" Ausdruck
```

Der Schleifeninhalt wird ausgeführt, bis die Bedingung (beliebiger Boolean-Ausdruck) hinter dem "UNTIL" wahr ist. Da die Bedingung am Schleifenende geprüft wird, wird die Schleife mindestens einmal durchlaufen.

2.9.6 WHILE-Schleife

Syntax:

```
"WHILE" Ausdruck "DO" Anweisung
```

Die Schleife wird durchlaufen, solange die Bedingung erfüllt ist.
Unterschiede zu "REPEAT":

- Die Bedingung (Boolean-Ausdruck) wird hier am Schleifenanfang geprüft. Dadurch wird die Schleife überhaupt nicht durchlaufen, wenn die Bedingung schon vor dem ersten Durchlauf "false" ist.
- Die Schleife bricht ab, sobald die Bedingung NICHT erfüllt ist (die REPEAT-Schleife wird durchlaufen, bis die Abbruchbedingung erfüllt ist).
- Der Schleifenkörper der WHILE-Schleife besteht formal nur aus einer einzigen Anweisung. In Form einer Verbundanweisung (mit "BEGIN" und "END" eingeklammert) kann man aber auch hier eine beliebig lange Anweisungsfolge benutzen.

2.9.7 FOR-Schleife

Syntax:

```
"FOR" Variable ":=" Startwert ("TO" | "DOWNTO") Endwert "DO" Anweisung
```

Die Variable wird auf den Startwert gesetzt und dann nach jedem Schleifendurchlauf auf den nächsthöheren (bzw. nächsttieferen, falls "DOWNTO" verwendet wird) Wert gesetzt, bis der Endwert überschritten (bzw. unterschritten) wird.

Als Schleifenzählvariable ist jede Variable eines geordneten Typs erlaubt, das sind:

- Ganzzahltypen (NICHT Real!)
- Char
- Boolean
- Aufzählungstypen
- Ausschnittstypen aus diesen genannten Typen.

Im Gegensatz zu einigen anderen PASCAL-Implementierungen muß die Variable nicht lokal sein.

Start- und Endwert sind beliebig komplizierte Ausdrücke, die zur Zählvariablen zuweisungskompatibel sein müssen.

Bitte denken Sie daran, daß hinter dem "DO" kein Semikolon stehen darf, es sei denn, Sie wollen ausdrücklich eine leere Schleife programmieren.

Der Rumpf der FOR-Schleife wird keinmal ausgeführt, wenn der Startwert echt größer (bei "TO") bzw. kleiner (bei "DOWNTO") als der Endwert ist.

Innerhalb der Schleife darf der Wert der Schleifenvariablen nicht verändert werden. Dies wird vom Compiler aber nicht überprüft! KICK-PASCAL berechnet vor dem ersten Schleifendurchlauf, wie oft die Schleife auszuführen und die Zählvariable weiterzuzählen ist.

Die Schleife

```
FOR v:= e1 TO e2 DO s;
```

entspricht in etwa:

```
v:= e1;
 count:= ord(e2) - ord(v);
 WHILE count >= 0 DO
   BEGIN
     s;                  { Schleifenrumpf ausführen }
     v:= succ (v);       { Zählvariable weitersetzen }
     count:= count - 1 { internen Zähler erniedrigen }
   END;
```

2.9.8 CASE-Anweisung

Syntax:

```
"CASE" Ausdruck "OF"
  { Konstantenliste ":" Anweisung }
  [ ("OTHERWISE" | "ELSE") [ Anweisung { ";" Anweisung } ] ]
"END"

Konstantenliste = Konstante [ ".." Konstante ] { "," Konstante [ ".."
Konstante ] }
```

Die CASE-Anweisung ist eine Mehrfach-Fallunterscheidung, bei der aufgrund des Vergleichs eines geordneten Ausdrucks (also Ganzzahlig, Boolean, Char, Aufzählungstyp) mit Konstanten verzweigt wird.

Wie Sie schon aus der Syntax erkennen können, gibt es hier zwei Ergänzungen zum ISO-Standard:

- Es können Bereiche von Konstanten angegeben werden.

  Beispiel:

```
Case i Of
  -MaxInt..0, 20..MaxInt:
          writeln('Eingabe nicht im erlaubten Bereich.');
  1..8:   writeln('Zu klein!');
  12..19: writeln('Zu groß!');
  9, 11:  writeln('Fast richtig.');
  10:     writeln('Richtig.')
End;
```

  Dieses CASE-Statement gibt zu jeder Integerzahl "i" einen Kommentar aus.

- Es gibt einen optionalen ELSE-Zweig, der wahlweise mit "ELSE" oder "OTHERWISE" eingeleitet wird. Dieser Teil kann beliebig viele Anweisungen enthalten.

  Beispiel:

```
Case c Of
  'A', 'a': writeln('Ah');
  'B', 'b': writeln('Bee');
  'C', 'c': writeln('Zeh');
  'D', 'd': writeln('Dee');
  '0'..'9': Begin
              writeln('Bitte keine Zahlen!');
              writeln('(Ich kann nämlich nicht zählen');
            End;
  'e'..'z': writeln('Irgend so'n Kleinbuchstabe.');
  'E'..'Z': writeln('Das war ein GROSSER Buchstabe.');
Else
  writeln;
  writeln('Tut mir leid, diesen Buchstaben kenne ich nicht.');
End;
```

Wie bereits erwähnt, hätte man statt ELSE auch OTHERWISE schreiben können. Ferner sehen Sie, daß vor ELSE (im Gegensatz zu IF-THEN-ELSE) ruhig ein Semikolon stehen darf.

Hat die CASE-Anweisung keinen ELSE-Zweig und der Ausdruck einen Wert, der in den Konstantenlisten nicht auftaucht, so wird ein Laufzeitfehler gemeldet. Es ist nicht erlaubt, einen Wert mehrfach in den Konstantenlisten aufzuführen. KICK-PASCAL meldet dabei aber keinen Fehler, sondern führt, wenn der Ausdruck einen doppelt vorgesehenen Wert hat, nur die erste Anweisung aus!

2.9.9 WITH-Anweisung

Syntax:

```
"WITH" Variable { "," Variable } "DO" Anweisung
```

Die Variable muß von einem RECORD-Typ sein. Innerhalb der WITH- Anweisung kann man dann jedes Feld dieses Records direkt wie eine Variable ansprechen.

Beispiel:

```
TYPE Person = RECORD
                Name: RECORD
                        Vor, Nach: STRING [40]
                      END;
                Plz: integer;
                Ort: STRING [50]
              END;

VAR P1, P2: Person;

...
WITH P1 DO
   BEGIN
     Name.Vor:= 'Dietrich';
     Name.Nach:= 'Gerstenberger';
     Plz:= 4799;
     Ort:= 'Borchen'
   END;
```

Die Anweisung ließe sich, da "Name" innerhalb der WITH-Anweisung ja eine eigenständige Variable ist, so weiter vereinfachen:

```
WITH P1 DO
   WITH Name DO
     BEGIN
       Vor:= 'Helmut';
       Nach:= 'Kohl';
       Plz:= 5300;
       Ort:= 'Bonn'
      END;
```

Dafür gibt es eine Abkürzung: Man faßt einfach die beiden WITH's zusammen:

```
WITH P1, Name DO
   BEGIN ...
   END;
```

Es ist aber nicht zwingend, daß die zweite Variable Bestandteil der ersten ist, es können auch voneinander unabhängige Variablen sein.

Beispiel:

```
WITH P1, P2.Name DO
   BEGIN
     Nach:= Name.Nach;
     P2.Plz:= Plz;
     P2.Ort:= Ort
   END;
```

Was geschieht hier? Durch "WITH P1" werden die Variablen "Name", "Plz" und "Ort" eingerichtet, die sich auf Bestandteile von "P1" beziehen, während "WITH P2.Name", die zu "P2" gehörigen Variablen "Vor" und "Nach" erzeugt. Also erhält in unserem Beispiel "P2" Nachname, Ort und Postleitzahl von "P1".

2.9.10 GOTO-Sprünge

Syntax:

```
"GOTO" Label
```

Ein Label kann (siehe Kapitel "Labeldeklaration") entweder eine Ziffernfolge oder ein Bezeichner sein.

Gemäß ISO-Standard kann man aus Strukturen mit "Goto" herausspringen (z. B. aus Schleifen heraus oder aus Unterprogrammen ins Hauptprogramm), aber nicht umgekehrt.

KICK-PASCAL macht sich aber nicht die Mühe, dies alles genau zu prüfen. So ist es z. B. möglich, in eine WHILE- oder REPEAT-Schleife hineinzuspringen.

Lediglich Einsprünge in FOR-Schleifen und WITH-Anweisungen werden abgefangen (und natürlich der Sprung in ein Unterprogramm - schon allein dadurch, daß ein dort gesetztes Label ja lokal deklariert sein muß).

Allgemein anerkannte Konvention über Goto's: Sprünge sollten nur da eingesetzt werden, wo der normale Programmablauf abgebrochen wird (z. B. wegen eines Eingabefehlers) oder wo das Programm ohne GOTO unnötigerweise übermäßig aufgebläht würde. Man sollte aber bei jedem GOTO einen Kommentar setzen, aus dem hervorgeht, warum und wohin hier gesprungen wird und bei jedem Label vermerken, von wo und unter welchen Bedingungen hierher gesprungen wurde.

REFERENZ

KAPITEL III

DIE DATENTYPEN VON KICK-PASCAL

III. Die Datentypen von KICK-PASCAL

3.1 Numerische Typen

KICK-PASCAL bietet nicht weniger als 7 verschiedene numerische Datentypen:

Integer
Der gewohnte Ganzzahlbereich -(MaxInt+1)..+MaxInt, MaxInt=32767

LongInt bzw. **Long**
32-Bit-Ganzzahltyp mit Vorzeichen, Bereich -(MaxLongInt+1) bis +MaxLongInt. Die Konstanten "MaxLongInt" und "MaxLong" haben den Wert 2147483647.

ShortInt bzw. **Short**
Vorzeichenbehafteter 8-Bit-Integertyp, MaxShortInt=MaxShort=127.

Cardinal bzw. **Word**
Dieser Ganzzahltyp ist vorzeichenlos, umfaßt also nur positive Werte und zwar im Bereich von 0 bis MaxCard=MaxWord=65535.

ShortCard bzw. **Byte**
Vorzeichenlose 8-Bit-Zahl, Wertebereich von 0 bis MaxShortCard=MaxByte=255.

Real
32-Bit-Realzahlen: 24 Bit Mantisse, 1 Bit Vorzeichen, 7 Bit Exponent. Das reicht für ca. 7-stellige Rechengenauigkeit im Bereich bis etwas über 8*10^18. Nicht gerade viel, aber für die meisten Zwecke genug.

LongReal bzw. **Double**
Seit Version 2.0 gibt es jetzt auch Fließpunktzahlen doppelter Genauigkeit nach IEEE-Norm. Sie sind 64 Bits breit (1 Bit Vorzeichen, 11 Bit Mantisse, 52 Bit+1 Hidden-Bit Mantisse), belegen im Speicher also 8 Bytes. Die Rechengenauigkeit ist etwa 15-stellig, der Exponentenbereich ist 1E+-308, das sollte genügen.

3.1.1 Ganzzahl-Typen

Fünf der genannten Zahlentypen, nämlich Integer, ShortInt, LongInt, Cardinal und ShortCard, sind sogenannte Ganzzahltypen, d. h. sie umfassen nur ganzzahlige Werte. Alle diese Typen können beliebig durcheinander verwendet werden, ohne daß man Typwandlungen vornehmen muß. Der Haken: KICK-PASCAL muß trotzdem dafür sorgen, daß verschiedene zusammen benutzte Datentypen aneinander angepaßt werden.

Diese Transformationen kosten teilweise nicht unerheblich viel Laufzeit, so daß es in der Regel besser ist, in Ausdrücken nur Daten gleichen Typs zu benutzen.

KICK-PASCAL wählt bei Kombination von unterschiedlichen Datentypen stets einen höheren, der beide umfaßt, z. B. wird die Summe eines Integer- und eines Cardinalwerts im LongInt-Bereich berechnet.

Ausschnittstypen sind stets Integer- oder LongInt-Typen, auch wenn sie theoretisch in einem "kleineren" Typ Platz hätten. Beispiel:

```
Type
  Hundert = 1..100; {belegt 16 Bit, d.h. ist Ausschnitt von Integer}
  NochMehr = 0..50000 { ist ein LongInt-Typ, obwohl auch Cardinal
möglich wäre. }
```

Es können auch Hexzahlen (Beispiel: $686b, $00C00276, $0d, $Bad) und Binärzahlen (Beispiel: %110, %101010101, %0011) verwendet werden.

Bei allen Ganzzahltypen gibt es folgende arithmetische Operatoren:

+	Addition
-	Subtraktion oder auch Vorzeichen
*	Multiplikation
DIV	Ganzzahlanteil der Division
MOD	Divisionsrest

Natürlich gilt "Punktrechnung vor Strichrechnung", so daß Multiplikation und Division Vorrang vor Addition und Subtraktion haben.

Gerechnet wird aber nur in den Wertebereichen Integer, Cardinal und LongInt. Wenn zwei ShortInt- oder ShortCard-Werte verknüpft werden, werden sie automatisch nach Integer oder Cardinal gewandelt.

Die beiden Divisionsoperatoren funktionieren nicht ganz korrekt, wenn einer der Operanden negativ ist: In der Mathematik ist es üblich, daß "a MOD b" stets ein Ergebnis zwischen 0 und dem Betrag von b-1 liefert und daß bei "a DIV b" immer das Ergebnis der normalen Division abgerundet wird.

Die Divisionsbefehle des Mikroprozessors MC68000, der in Ihrem AMIGA steckt, arbeiten aber anders: Hier wird erst mit den Beträgen der Operanden gerechnet, und anschließend wird das Vorzeichen des Ergebnisses negativ gewählt, wenn genau einer der Operanden negativ war. Aus Geschwindigkeitsgründen muß auch KICK-PASCAL diese eigentlich falschen Operationen benutzen.

Die folgende Tabelle soll diesen Sachverhalt verdeutlichen:

		Mathematisch	MC 68000
42	DIV 10	4	4
(-42)	DIV 10	-5	-4
42	DIV (-10)	-5	-4
(-42)	DIV (-10)	4	4
42	MOD 10	2	2
(-42)	MOD 10	8	-2
42	MOD (-10)	8	-2
(-42)	MOD (-10)	2	2

So tragisch, wie es zunächst scheint, ist das Ganze aber nicht, denn in der Praxis ist der linke Operand selten und der rechte fast nie negativ. Außerdem ist sowohl bei der mathematisch üblichen als auch der von Motorola (dem 68000-Hersteller) vorgenommenen Definition gewährleistet, daß stets folgende Gleichung gilt:

```
a = b*(a DIV b) + (a MOD b)
```

Bei arithmetischen Operatoren kann es natürlich zu Überläufen kommen, wenn das Ergebnis einer Rechnung nicht im gewählten Rechenbereich liegt. Es kommt nur dann zu einer Fehlermeldung, wenn die entsprechende Compiler-Option mit "{$opt a+}", dem Pull-Down-Menü "Optionen/Compiler" oder dem "Preferences"-Tastaturmenü eingeschaltet wurde, andernfalls ist das Ergebnis undefiniert. Genaueres über diese Option erfahren Sie im Kapitel VI, Abschnitt 1.5: "Compiler-Optionen". Im Abschnitt "Typkonvertierungen" dieses Kapitels finden Sie Tips, wie Sie in bestimmten Fällen Überläufe vermeiden können.

Neben den erwähnten arithmetischen gibt es noch die logischen Operatoren AND, OR, XOR, NOT, SHL und SHR, die Zahlen bitweise verknüpfen:

AND, OR und XOR funktionieren im Prinzip genau wie die gleichnamigen Boolean-Operatoren, nur daß hier jeweils die Bits der Operanden zu den entsprechenden Bits des Ergebnisses verknüpft werden.

Hier sind einige Beispiele - mit Binärzahlen, da es hier ja auf die Bits ankommt:

```
    %11111100
AND %10111010
------------
=   %10111000
```

Ein Bit wird gesetzt, wenn beide Bits der Operanden gesetzt sind.

```
    %1100
OR  %10001010
------------
=   %10001110
```

Ein Bit wird gesetzt, wenn mindestens eins der Operandenbits gesetzt ist.

```
    %11001100
XOR %11111010
------------
=   %00110110
```

Ein Bit wird gesetzt, wenn die beiden Operandenbits verschieden sind, also genau eins gesetzt ist.

NOT führt auf ganzen Zahlen eine logische Negation aus, d. h. alle Bits des folgenden Operanden werden invertiert.

Beispiele:

```
NOT 0       = -1
NOT %110    = %1111111111111001 = -7
NOT 26730   = -26731
```

Genau wie bei booleschen Ausdrücken hat NOT auch hier höchste Priorität, so daß

```
NOT a*b          äquivalent zu
(NOT a) * b      ist.
```

Außerdem gibt es noch die Operatoren SHL und SHR, "Shift left" und "Shift right". Sie verschieben die Bits des linken Operanden um so viele Positionen nach links bzw. rechts, wie der zweite Operand angibt. Das entspricht einer Multiplikation mit bzw. einer Division durch eine Zweierpotenz.

Beispiele:

```
1 Shl 6       = 64    (= 1 * 2^6)
10 Shr 1      = 5      (= 10 div 2^1)
MaxInt Shr 14       = 1      (= (2^15-1) div (2^14))
```

Beide Operanden müssen ganzzahlig sein. Der rechte Operand darf nicht negativ sein, andernfalls ist das Ergebnis undefiniert (wenn es sich um eine negative Konstante handelt, z. B. "17 shl -2", meldet bereits der Compiler einen Fehler). Es findet kein Test auf Überlauf statt, auch nicht, wenn die Compiler-Option "Arithm. Überl." aktiv ist.

Die Shift-Operanden zählen zu den Multiplikationsoperanden, sie haben also dieselbe Priorität wie "*", "DIV" oder "MOD".

Beispiel:

```
NOT 1 shl 5 + 2    entspricht    ((NOT 1) shl 5) + 2
```

Hier ist noch einmal eine Übersicht über alle Ganzzahl-Operatoren und ihre Auswertungsreihenfolge. Es werden immer zuerst die Operatoren in den oberen und dann die in den unteren Zeilen der Tabelle zusammengefaßt. Operatoren, die in der selben Zeile stehen, werden von links nach rechts ausgewertet.

	Arithmetisch	**Logisch**
Invertierung:		NOT
Multiplikativ:	*, DIV, MOD	AND, SHL, SHR
Vorzeichen:	+, -	
Additiv:	+, -	OR, XOR
Vergleiche:	<, >, <=, >=, =, <>	

Bei der Ausgabe mit "Write" werden Integer-Zahlen linksbündig ausgegeben (Abweichung vom Standard!).

```
Write(17, 4, 17+4);
```

erzeugt also die Ausgabe

```
17421
```

Sie erhalten eine rechtsbündige Ausgabe, indem Sie hinter der Zahl die gewünschte Feldbreite als Ganzzahl-Ausdruck angeben:

```
Write(17:5, 4:5, 17+4:10)
```

gibt aus:

```
   17    4         21
```

Bei der Eingabe mittels "Read" werden nur dezimale Zahlen akzeptiert. Im Falle einer unerlaubten Eingabe wird aber kein Laufzeitfehler gemeldet.

3.1.2 Gleitpunktzahlen

KICK-PASCAL benutzt die normalen Gleitpunkt-Bibliotheken, die das Betriebssystem zur Verfügung stellt. Insbesondere stehen zwei verschiedene Zahlenformate zur Verfügung: Die einfach genauen "Real"-Zahlen und ein doppelt genaues Format, das "LongReal" oder "Double" heißt.

Eine Real-Zahl ist 32 Bit "breit" und belegt also 4 Bytes im Speicher, während Double 64 Bit=8 Byte breit ist. Rechnungen mit doppelter Genauigkeit sind erheblich langsamer als normale Real-Rechnungen, allerdings unterstützt die Mathebibliothek doppelter Genauigkeit eine evtl. vorhandene FPU (Fließkomma-Coprozessor).

Es ist erlaubt, in Ausdrücken beide Zahlenformate zu kombinieren oder auch Gleitpunktzahlen mit Ganzzahlausdrücken zu verknüpfen. Auch hier gilt, daß Typumwandlungen nicht unerheblich Rechenzeit benötigen.

Auf beiden Zahlentypen gibt es folgende Operatoren:

*	Multiplikation
/	Division
+	Addition
-	Subtraktion oder Vorzeichen

Natürlich gilt auch hier "Punkt- vor Strichrechnung". Ferner gibt es die üblichen Vergleichsoperationen, die einen Boolean-Wert liefern:

=, <>, <, >=, >, <=

Ein Problem der meisten Programmiersprachen ist die "Überladung" von Operatoren: "*", "+" und "-" sowie die Benutzung der Vergleichsoperatoren sowohl für Ganzzahl- als auch für Realoperationen. Dabei gelten folgende Regeln:

- Wenn einer der beiden Operanden von einen Fließkommatyp ist, wird auch im Fließkommabereich gerechnet.
- Bei einer Verknüpfung eines Real- und eines Double-Operanden wird das Ergebnis im Double-Bereich berechnet.

Der Divisionsoperator "/" liefert stets eine Fließkommazahl als Ergebnis, bei "DIV" und "MOD" sind nur ganze Zahlen als Operanden erlaubt.

Die beiden Fließkommatypen sind auch wertzuweisungskompatibel. Wenn man aber einen Real-Wert an eine Double-Variable zuweist, kann man Überaschungen erleben:

```
Var r: Real;
    d: Double;
Begin
  r:= 0.2;
  d:= r;
  writeln(r, d)
End.
```

Ergebnis: `0.2  0.1999999880079071`

Die Double-Variable hat also scheinbar einen ungenauen Wert. Wie ist das möglich, da Double doch, Gerüchten zufolge, genauer ist als Real? Nun, eine Zahl wie 0.2 ist binär nicht mit endlich vielen Stellen darstellbar (im Dezimalsystem kann man 1/3 ja auch nicht als endlichen Dezimalbruch darstellen). Wenn diese Zahl nun an die Real-Variable zugewiesen werden soll, erhält die Variable in Wirklichkeit den besten darstellbaren Näherungswert für 0.2 - und das ist in dezimaler Darstellung etwa 0.1999999880079071, Sie sehen hier die etwa 7stellige Genauigkeit. Wenn diese Realzahl nun ausgegeben wird, berücksichtigt die Ausgaberoutine des KICK-PASCAL-Laufzeitsystems die geringe Rechengenauigkeit und rundet auf 6 oder 7 Dezimalstellen, je nach gewünschtem Ausgabeformat. Also wird 0.2 ausgegeben, und der Anwender wird mit diesem kleinen Trick über die geringe Rechengenauigkeit hinweggetäuscht.

Durch "d:= r" bekommt aber die Double-Variable "d" den ungenauen Näherungswert zugewiesen - und bei der nachfolgenden Ausgabe wird dann nicht auf 6, sondern auf 15 Dezimalstellen gerundet. So führt die höhere Genauigkeit zu einem scheinbar ungenaueren Ergebnis.

Bei der Ausgabe mittels "Write" gibt es mehrere Optionen:

- `write(x)` oder `write(x:b)` oder `write(x:b:0)`

  Im Bereich 10000 < x <= 0.1 wird die Zahl x "normal" ausgegeben, andernfalls in Exponentialdarstellung. In beiden Fällen werden überflüssige Nullen am Ende weggelassen.

  "b" gibt hier wie immer die Breite des Feldes an, in das die Zahl rechtsbündig ausgegeben werden soll.

  Bei negativen Zahlen beginnt die Ausgabe mit dem Minuszeichen, bei positiven Werten wird am Anfang ein Leerzeichen ausgegeben. Beispiele:

```
write(-4/2)              -2
write(pi)                 3.14159265358979
write(26.731e+9:20)       2.6731E+10
```

- `write(x:b:n)  mit n > 0`

  Die Zahl x wird in Festkommadarstellung mit n Nachkommastellen ausgegeben. Beispiele:

```
write(17.4:12:3)          17.400
write(26.731: 12: 2)      26.73
write(pi:25:25)           3.1415926535897930000000000
```

  Bei "Real" sind bis zu 20 Stellen möglich. Beachten Sie aber, daß der AMIGA hier nur mit etwa 6 1/2 - stelliger Genauigkeit rechnet, so daß es schon an der 7. Stelle (Vorkommastellen mitgerechnet) zu Rundungsfehlern kommen kann und der ganze Rest nur mit Nullen aufgefüllt wird. Im "Double"-Bereich sind bis zu 25 Nachkommastellen möglich, aber auch hier werden maximal 16 "echte" Stellen ausgegeben, der Rest ist Null.

- `write(x:b:n)  mit n < 0`

  Die Zahl x wird in Exponentialdarstellung und in "-n" - stelliger Genauigkeit ausgegeben. Beispiele:

```
write(26.731:12:-4)       2.673E+01
write(0.07:12:-5)         7.0000E-02
write(0:12: -3)           0.00E+00
```

Die ganzen mathematischen Funktionen, wie "SIN", "EXP", "SQRT" oder "LN", sind nur im Real-Bereich definiert. Sie können zwar wegen der beidseitigen Wertzuweisungskompatibilität auch auf Zahlen doppelter Genauigkeit angewandt werden, liefern dann aber ungenaue Ergebnisse, nämlich in der siebenstelligen Real-Genauigkeit! In der "mathieeedoubtrans.library" finden Sie alle nötigen Funktionen in doppelter Rechengenauigkeit.

3.2 Boolean

Der Datentyp "Boolean" umfaßt die beiden Werte "True" und "False". Sie sind geordnet:

Es gilt

```
False < True
ord(False) = 0
ord(True) = 1
```

Eine Variable dieses Typs belegt ein Byte. Es gibt folgende Verknüpfungen:

`AND`	logisches Und
`OR`	logisches Oder
`XOR`	logisches auschließliches Oder

OR und XOR haben gleiche Priorität und zwar eine kleinere als AND.

Beispiel:

```
a OR b AND c XOR d
```

entspricht:

```
a OR (b AND c) XOR d
```

und damit (gleiche Priorität von OR und XOR, deshalb Auswertung von links nach rechts):

```
(a OR (b AND c)) XOR d
```

Außerdem gibt es noch die Negation NOT, die höchste Priorität hat:

`NOT a AND b` entspricht deshalb: `(NOT a) AND b.`

Umsteiger von BASIC (Was ist das? Nie gehört!) nach PASCAL wissen oft nicht, daß die Booleschen Verknüpfungen in PASCAL dieselbe Priorität wie arithmetische Operatoren haben. Somit wäre

```
IF x>=0 AND x<10 THEN ...
```

falsch: "AND" hat Vorrang vor den Vergleichoperatoren, so daß zuerst "0 AND x" zusammengefaßt würde (in Standardpascal nicht erlaubt, bei KICK-PASCAL ist dies, falls x eine Ganzzahl-Variable ist, eine binäre Verknüpfung, die stets den Wert 0 hat), und übrig bliebe der unsinnige und syntaktisch falsche Mehrfachvergleich

```
IF x >= (0 AND x) < 10 THEN ...
```

Richtig wäre:

```
IF (x>=0) AND (x<10) THEN ...
```

Boolean-Werte können mit "Write" ausgegeben werden.

Beispiel:

```
write(1=2)
```

gibt "false" aus.

3.3 Char

Ein "Char" ist ein einzelnes Zeichen ("Character"). Es wird wahlweise mit einfachen oder doppelten Anführungszeichen eingeschlossen.

Folgendes wären also korrekte Char-Konstanten:

'A' "A" 'x' '\' '2' """

Das letzte Beispiel hätte man auch anders darstellen können: Ein Hochkomma wird in Hochkommata eingeschlossen, indem man es doppelt setzt, also '' '. Standard-PASCAL erlaubt es nicht, ein Char mit doppelten Hochkomma einzuschließen, so daß dort die letztere die einzige zulässige Darstellung des Apostrophs wäre.

KICK-PASCAL verwendet natürlich den normalen 8-Bit-Amiga-Zeichencode. Nicht-schreibbare Zeichen sind die Codes von 0 bis 31 sowie von 128 bis 159. Dafür gibt es verschiedene Literalarten: Zum einen wäre da die gewöhnliche Schreibweise "Chr(n)", wobei "n" ein ganzzahliger Ausdruck im Bereich von 0 bis 255 ist. Falls der Parameter konstant ist, kann die Chr-Funktion auch als Konstante verwendet werden, d. h.

`"CONST Linefeed = chr(10)"` oder `"CASE c OF chr(13): ..."`

sind möglich.

Ferner akzeptiert KICK-PASCAL die von gewissen unter MS-DOS laufenden PASCAL-Implementierungen her bekannte Schreibweise "#n", wobei n eine ganzzahlige Konstante im zulässigen Bereich ist. Und last not least kann man (da Charkonstanten äußerlich Ein-Zeichen-Strings sind) wie bei Stringkonstanten verfahren, z.B. "\10 oder "$0a als Ersatz für chr(10). Näheres dazu finden Sie im String-Kapitel.

3.4 Aufzählungs- und Ausschnittstypen

PASCAL bietet Ihnen die Möglichkeit, sich Datentypen selbst zu definieren. Eine einfache Möglichkeit sind die Aufzählungstypen.

Man erzeugt diesen Datentyp ganz einfach, indem man die Werte, die jeweils durch einen Bezeichner repräsentiert werden, als Liste in runden Klammern aufzählt. Ein beliebtes Beispiel:

```
TYPE Wochentag =    (Montag, Dienstag, Mittwoch, Donnerstag,
                     Freitag, Samstag, Sonntag);
```

Diese Zeile definiert nicht nur den neuen Datentypen "Wochentag", sondern auch die sieben Konstanten "Montag" bis "Sonntag". Deshalb dürfen diese Bezeichner zuvor noch nicht definiert sein.

"Wochentag" ist ein Datentyp, der die sieben diskreten Werte "Montag" bis "Sonntag" umfaßt. "Mittwoch" z. B. ist eine Konstante mit dem Wert "Mittwoch" - klingt seltsam, ist aber so.

Welche Eigenschaften hat ein solcher Datentyp, und was kann man damit machen? Die Elemente sind geordnet: Es gilt Montag < Dienstag < Mittwoch < ... < Sonntag. Des weiteren haben sie auch Ordnungszahlen: Ord(Montag)=0, Ord(Dienstag)=1 usw. Auch die Funktionen "Succ" und "Pred" können auf Aufzählungstypen angewendet werden, z. B. Succ(Montag)=Dienstag, Pred(Freitag)=Donnerstag. Das sind dann aber auch so ziemlich alle Operationen auf Aufzählungstypen. Ihr Sinn ist im wesentlichen, Programme übersichtlicher zu gestalten, z. B. indem man Wochentage nicht durchnumeriert, sondern wie im obigen Beispiel über ihre Namen verwaltet. Eine Variable eines Aufzählungstyps belegt übrigens immer 2 Bytes.

Die andere Klasse von einfachen selbstdefinierten Datentypen sind die Ausschnittstypen. Dazu nimmt man sich einen bereits existierenden geordneten Datentypen, also einen Ganzzahltypen, Boolean, Char oder einen Aufzählungstypen und schränkt seinen Wertebereich auf ein beliebiges Interval ein. Das folgende Programmfragment enthält einige Ausschnittstypen:

```
TYPE
   Positiv    =  1..MaxInt;  { Dies ist ein Ausschnitt aus Integer,
                               der alle positiven Integerzahlen umfaßt }
   Wochentag  =  (Montag, Dienstag, Mittwoch, Donnerstag, Freitag,
                 Samstag, Sonntag);  { kennen wir schon }
   Arbeitstag=   Montag..Freitag; { ein Ausschnitt aus dem Typen
                                    "Wochentag" }
   Gross      =  'A' .. 'Z'; { 26-elementiger Ausschnittstyp von
                               "Char" }
   Ziffern    =  '0' .. '9'; { auch dies ist ein Char-Ausschnitt }

VAR n:              Positiv; { die Variable "n" kann nur positive Werte
                               haben }
    i:              0 .. 2673142; { ein Ausschnitt aus "LongInt" }
    PlusMinusCard: -MaxCard .. +MaxCard; { auch dies ist ein Ausschnitt
                                           aus "LongInt" }
    Seltsam:        False .. True; { umständliche Art, eine Boolean-
                                     Variable zu deklarieren }
    Heute:          Wochentag;
    IstLeider:      Montag..Freitag;  { ist im Vergleich zu "Heute"
                                        auf 5 Werte beschränkt }
    HätteGern:      Samstag..Sonntag; { Achtung: beliebter Fehler ist,
                    hier "(Samstag,Sonntag)" zu schreiben. Überlegen
                    Sie doch mal, warum das nicht geht! }
```

Man definiert Ausschnittstypen also, indem man eine Unter- und eine Obergrenze angibt. Beide Grenzen müssen Konstanten sein; die untere muß echt kleiner als die obere sein, denn sonst meldet der Compiler einen Fehler.

Eine Variable eines Ausschnittstyps verhält sich exakt wie eine des Grundtyps. Einziger Unterschied: Bei jeder Wertzuweisung wird geprüft, ob die Variable im erlaubten Bereich liegt; andernfalls bricht das Programm mit einer Fehlermeldung ab.

3.5 Stringtypen

Ein String ist eine beliebig lange Folge von Zeichen. Es gibt zwei verschiedene Stringtypen:

- Die Deklaration "String[n]" ist identisch mit "Array [1..n] of Char". Das Symbol "String" ohne Längenangabe entspricht "String[80]". Zwar ist "String" kein Typbezeichner, sondern eine sog. Directive. Seit Version 1.5 von KICK-PASCAL kann "String" aber überall als Pseudo-Typbezeichner verwendet werden, z. B. auch als Typ in Parameterlisten.
- Der Datentyp "Str" ist ein Zeiger auf einen String.

Beim ersten Typ müssen Sie eine maximale Länge festlegen. Hier können Sie über den Index auch jedes Zeichen einzeln ansprechen. Das Ende wird mit einem Zeichen chr(0) markiert. Weil auch dieses Zeichen ein Byte benötigt, können Sie in einem "String[5]" effektiv höchstens 4 Zeichen unterbringen.

"Str" ist nichts weiter als ein Pointer. Das AMIGA-Betriebssystem benötigt häufig solche Zeiger auf Texte, und das ist auch der einzige Grund, warum dieser Typ in KICK-PASCAL implementiert wurde.

Er ist nämlich eine etwas haarige Angelegenheit. Betrachten Sie bitte einmal folgendes Programm:

```
Program strings;
Var s1:str; s2:String[10];
Begin
 s2:='Hallo';
 s1:=s2;
 s2:='Horrido';
 writeln(s1)
End.
```

Raten Sie mal, welche Ausgabe dieses Programm macht: Zuerst wird in "s2" der Text 'Hallo' abgelegt (s2[1]='H', ... , s2[5]='o', s2[6]=chr(0)). In der nächsten Zeile wird "s2" an "s1" zugewiesen.

"s1" besitzt aber keinen eigenen Speicher außer den 4 Bytes, die ein Pointer benötigt ("s2" belegt 10 Bytes). Vielmehr zeigt der ZEIGER "s1" nach dieser Zeile auf die Variable "s2". Wenn dann in der nächsten Zeile ein anderer Text an "s2" zugewiesen wird, ändert sich auch der "Inhalt" von "s1". Ergo gibt das Programm 'Horrido' aus - ein eklatanter Verstoß gegen die Prinzipien gepflegter PASCAL-Programmierung.

An diesem Beispiel sehen Sie auch, daß Sie den Wert, auf den "s1" zeigt, erhalten, ohne (wie bei normalen Pointern notwendig) "s1^" zu schreiben. Das liegt daran, daß KICK-PASCAL Strings intern ohnehin nur über ihre Speicheradresse verwaltet werden.

Gefährlich wird es (und auch das dürfte es in PASCAL eigentlich nicht geben), wenn im obigen Beispiel "s2" lokal in einer Procedure und "s1" als globale Variable

deklariert wäre. In der Prozedur wird "s1" zunächst ein Zeiger auf die lokale Variable "s2" zugewiesen. Wenn das Programm dann die Prozedur verläßt, existiert die Variable "s2" garnicht mehr, und "s1" zeigt deshalb irgendwo in die Wallachei. Sobald der entsprechende Speicherbereich wieder benutzt wird, wird der Wert von "s1" geändert.

Dieser Exkurs zeigt, daß Sie den Str-Typ hauptsächlich im Zusammenhang mit der AMIGA-Betriebssystemprogrammierung verwenden sollten. Er ist auch ohnehin nicht so interessant, weil man eine solche Variable weder über Read/ReadLn einlesen noch die Zeichen einzeln ansprechen kann. Alles, was Sie mit "str" anfangen können, ist:

- eine Variable zuweisen.
- eine Stringkonstante zuweisen, z.B. "s1:='Halleluja'". Eine solche Zuweisung kann bedenkenlos auch in einer Prozedur niedrigeren Ranges erfolgen, denn die Konstante liegt nicht im Variablenspeicher, sondern im Programmcode.
- mit anderen Strings vergleichen.
- mit Write(ln) ausgeben.
- als Parameter in Funktionsaufrufen verwenden.

Damit wären wir auch schon beim nächsten Thema:Operationen auf Strings.

KICK-PASCAL hat folgendes auf Lager:

- Wertzuweisungen - wie oben gesehen.
- Ausgabe mit Write/WriteLn.
- Eingabe mit Read/ReadLn (geht nicht mit Str).
- Vergleichsoperationen =, <>, <, <=, >, >=. Es kann auch ein String mit einem Char verglichen werden.
- Verbinden mit "+" oder der "Concat" - Funktion - Mit "Copy" auf Ausschnitte zugreifen.

Stringkonstanten können wahlweise mit ' ' (wie in Standard-PASCAL) oder mit " " eingeschlossen werden. Es gibt die Möglichkeit, Steuerzeichen in Strings einzusetzen: Man unterbricht den String mit ' oder " - aber mit dem selben Zeichen, mit dem man ihn begonnen hat - und kann dann entweder mit $xx eine Hexadezimalzahl oder mit \ eine Dezimalzahl einschieben.

Beispiel:

```
Program Steuercodes;
Begin
  writeln('xxx aaa'$0d'bbb');
  { $0d oder dezimal 13 entspricht dem Code "CR" ("Wagenrücklauf").
  Der Cursor springt an dieser Stelle an den Zeilenanfang, so daß
  "xxx" von "bbb" überschrieben wird. }

  writeln("Zeile 1"\10$a"Zeile 2");
  { Dezimal 10 = Hex $a entspricht "LF" ("Linefeed"/"Zeilenvor-
  schub"). Hier werden zwei Zeilenenden eingeschoben. }
```

```
    writeln(''$9b'33mJetzt '\155'32;4mwird''s'\155'31;0;3m bunt!''$0a);
    { Der Steuercode dez 155 bzw. Hex $9b heißt "Escape" oder "CSI"
    ("Control Sequence Introducer" - "Steuersequenzeinleiter"). Damit
    haben Sie enorm viele Möglichkeiten, die ich hier nicht einzeln
    aufzählen kann. Setzt man z. B. dahinter eine oder mehrere
    Dezimalzahlen (ggf. durch ";" getrennt) und beschließt die Sequenz
    mit einem kleinen "m", wird das Aussehen des Textes beeinflußt: die
    Zahlen 0..7 stellen verschiedene Textattribute ein (unterstrichen,
    kursiv, ...) und Zahlen 30..37 setzen die Schriftfarbe.
    Außerdem fällt auf, daß
    a) der String mit einem Steuercode beginnt, indem der Leerstring ''
       vorangestellt wird,
    b) ein Apostroph ' im String enthalten ist, indem er doppelt
       gesetzt wird und
    c) die Zeichenkette mit einem Steuerzeichen endet, ohne daß
       dahinter noch ein Hochkomma stehen muß. }

    writeln('Zeile 4'\n'Zeile'\e'33;0m 5');
    { Hinter "\" können Sie auch die Buchstaben "n" und "e" setzen.
    Ersterer entspricht dem Zeichen LF (Code $0a), letzterer "CSI"
    (Code $9b). }

End.
```

Problematisch wird es, wenn auf oben beschriebene Weise ein Nullbyte in einen String eingesetzt wird. Sowohl KICK-PASCAL als auch das AMIGA-Betriebssystem betrachten dies in der Regel als Stringendemarkierung, so daß der Rest eventuell verloren geht. Wenn Sie einmal eine solche Null benötigen - z.B. bei der Intuition-Funktion "DisplayAlert" - müssen Sie den String äußerst vorsichtig behandeln.

Wenn ein String mit einem Steuerzeichen beginnen soll, können Sie am Anfang statt des Leerstrings auch das Zeichen "#" verwenden. Außerdem kann "#" überall als Ersatz für den Backslash "\" verwendet werden. Eine Zeile aus dem obigen Beispiel könnte also auch so aussehen:

```
write(#$9b'33mJetzt '#155'32;4mwird''s'#155'31;0;3m bunt!''$0a);
```

Durch solche Steuercodes kann ein String ganz schön lang werden. Er darf aber normalerweise das Zeilenende nicht überschreiten. Zu diesem Zweck gibt es die Möglichkeit, die beiden Zeichen "\&" in den String einzuschieben. Dahinter können dann beliebig viele Zeilenenden, Leerzeichen oder sogar Kommentare stehen, bevor der String fortgesetzt wird.

Beispiel:

```
write('Ein Gedicht:'\n\n\e'32mvon Douglas Adams'\n\n\&
    \x'[33mO zerfrettelter Grunzwanzling!''\n\&
    'Dein Harngedränge ist für mich'\n\&
    'Wie Schnatterfleck auf Bienenstich.'\&
    \n\n\c'31mDen Rest erspare ich Ihnen.')
```

Dieser Befehl gibt einen mehrere Zeilen langen Text auf einen Schlag aus. Allerdings ist in KICK-PASCAL die Länge von String-Konstanten auf 400 Zeichen

beschränkt - um einen Roman auszugeben, werden Sie also doch mehr als eine Zeichenkette benötigen.

Im obigen Beispiel tauchen außerdem die Einschübe "\c" und "\x" auf.

"\x" entspricht dem Escape-Zeichen #$1b. Zusammen mit einer folgenden eckigen Klammer "[", ist es äquivalent zum CSI-Zeichen. Diese Folge hat aber den Vorteil, daß sie auch vom Druckertreiber verstanden wird. Man kann \x'[' mit \c abkürzen.

Es gibt noch zwei wichtige Funktionen zur Stringbehandlung: "Concat" und "Copy":

Concat(string1, string2, ...)

hängt die übergebenen Strings zusammen. Das Ergebnis wird als "temporärer String" zurückgegeben.

"Was ist das?" Die Variablen eines "String[N]"-Typs besitzen jeweils einen für sie reservierten (N Bytes langen) Speicherbereich. Auch eine Stringkonstante liegt an einer bestimmten Stelle im Speicher, und ein "Str"-Zeiger zeigt wenigstens auf eine Speicheradresse. Das Ergebnis, das "Concat" zurückgibt, hat aber keinen vorher bestimmten Speicherplatz, vielmehr wird unmittelbar vor dem Zusammenhängen irgendwo die benötigte Menge Speicher reserviert (denn das Ergebnis kann ja beliebig lang werden) und muß nachher wieder freigegeben werden. Der String ist also quasi nach dem Lesen zu vernichten.

(Wußten Sie schon: Dokumente der höchsten NATO-Geheimhaltungsstufe tragen den Stempel "DBR" - "Destroy Before Reading"....) Von der internen Verwaltung dieser Strings merken Sie als PASCAL-Programmierer in der Regel nichts, denn wenn Sie einen "Concatinierten" String mit Write ausgeben oder an eine Stringvariable zuweisen, managed das PASCAL-Laufzeitsystem das Ganze für Sie.

Eine Einschränkung gibt es aber: Ein temporärer String kann nicht an eine "Str"-Variable zugewiesen werden. Grund: Ein temporärer String ist ja nur "von begrenzter Lebensdauer", und eine Str-Variable hat ja auch keinen Speicher, in dem sie den String aufbewahren könnte, sondern kann lediglich auf eine Zeichenfolge zeigen. Und ein Zeiger auf einen String, der sich bald in Wohlgefallen auflöst, wäre ja wenig sinnvoll.

Deshalb ist es natürlich auch nicht erlaubt, solche Strings als Parameter an Prozeduren und Funktionen zu übergeben, die einen "Str"-Parameter erwarten. Dazu gehören z. B. die PASCAL-Standardprozedur "Assign(datei, namestr)", die Funktion "Pos(str1, str2)" und jede Menge AMIGA-Systemfunktionen. Sie müssen hier also bei Bedarf das "Concat"-Ergebnis zunächst an eine Stringvariable zuweisen. Übrigens: Sie können Strings ersatzweise auch mit "+" aneinanderhängen.

Beispiel:

```
s := 'Hallo'+'Du'+Concat(Concat('Alter','Eimer','Wie)+'Geht's', '?')
```

Danach hat s den Wert "HalloDuAlterEimerWieGeht's?"

Copy(string, pos, len)

Diese Funktion liefert einen "len" Zeichen langen Ausschnitt ab Position "pos" aus der Zeichenkette "string". Auch hier ist das Ergebnis ein temporärer String, es gilt also sinngemäß dasselbe wie bei der Konkatination (lachen Sie nicht, dieses Wort gibt's wirklich).

Beispiel:

```
write(Copy('PASCAL the best - .... the rest!',8,3))
```

gibt "the" aus.

Wahrscheinlich war die erste Programmiersprache, die Sie gelernt haben, BASIC, so daß Sie es wahrscheinlich gewohnt sind, mit "mid$(a$,17,1)" auf das 17te Zeichen von "a$" zuzugreifen. Trotzdem sollten Sie (so schwer das auch oft fällt) mit dieser schlechten Gewohnheit brechen und stets a[17] statt Copy(a,17,1)) schreiben (Vorausgesetzt, a ist eine Variable eines String[x]-Typs). Dafür gibt es gleich vier gute Gründe:

1. Ersteres ist kürzer.
2. Es geht deutlich schneller.
3. Außerdem ist es Standard-PASCAL-kompatibel.
4. "Copy" gibt stets einen String zurück, so daß Sie das Ergebnis nicht an eine Char-Variable zuweisen können, auch wenn es nur ein Zeichen lang ist.

Bei einem Stringausdruck, einer Zeichenkettenkonstanten oder einer "Str"-Variablen ist es nicht möglich, wie oben beschrieben, auf ein einzelnes Zeichen zuzugreifen. Dafür gibt es - als besonderes KICK-PASCAL-Feature ab Version 1.5 - eine andere Schreibweise: hinter jeden Zeichenkettenausdruck kann ein Punkt, gefolgt von einem Pseudo-Index, gesetzt werden.

Ein Beispiel:

```
Program streawkceuR; Var Vorname, Nachname: String;
    i: integer;
Begin
  write('Vorname:  '); readln(Vorname);
  write('Nachname: '); readln(Nachname);
  For i:=12 Downto 1 Do
    write('Ihr Name ist'.[i]);
  writeln(':')
  For i:=Length(Vorname)+Length(Nachname)+1 Downto 1 Do
    write((Vorname+' '+Nachname).[i])
End.
```

Dieses Prgrämmchen fragt den User nach seinem Namen und gibt dann erst den Text "Ihr Name ist" und dann den vollständigen Namen rückwärts aus, indem es die beiden Stringausdrücke zeichenweise von hinten nach vorn schreibt.

Was wäre in jenem Programm der Unterschied zwischen "Vorname[5]" und "Vorname.[5]"? Ersteres ist eine Char-Variable, nämlich das fünfte Element des

Char-Arrays "Vorname". Dieser Variablen kann man z. B. auch einen Wert zuweisen. Das zweite ist ein Char-Ausdruck, so wie "Chr(k+8)" oder "Succ('A')", und kann deshalb nicht auf der linken Seite einer Wertzuweisung auftreten. "Stringausdruck.[Index]" ist also identisch mit "Copy(Stringausdruck,Index,1)", nur daß ersteres ein Char und letzteres einen String liefert.

3.6 Pointertypen

Außer den normalen Pointern besitzt KICK-PASCAL noch einen typfreien Pointertypen namens "ptr". Eine Variable dieses Typs kann auf einen beliebigen Typ zeigen - weshalb es auch nicht möglich ist, mit "^" diese Variable anzusprechen. Natürlich kann auf einen solchen Zeiger auch nicht New oder Dispose angewandt werden, denn es ist ja gar nicht klar, wie viel Speicher dabei reserviert bzw. freigegeben werden muß.

Wahrscheinlich werden Sie sich jetzt fragen, welchen Sinn dann ein solcher Zeiger hat. Nun, im Gegensatz zu normalen Zeigern kann an eine Ptr-Variable ein beliebiger Pointertyp zugewiesen werden und umgekehrt. Ein Beispiel:

```
Program Pointer;
 Var p1:^Typ1; p2:^Typ2; p:ptr;
 Begin
   ... p:=p1; p2:=p; ...
```

Das Ergebnis dieser Zuweisungen: p2 zeigt auf dieselbe Variable wie p1, obwohl diese eigentlich verschiedene Typen haben (im nächsten Abschnitt werden Sie erfahren, wie so etwas auch einfacher geht).

Jetzt werden Sie wahrscheinlich denken: OK, aber welchen Sinn hat SO ETWAS denn nun wieder? Darauf sei ebenso kurz wie ausweichend geantwortet, daß Sie das zur System-Programmierung benötigen, worauf in einem der nächsten Kapitel noch eingegangen wird.

NEW und DISPOSE verwenden übrigens nicht den reservierten Arbeitsspeicher, sondern die Exec-Funktionen "Allocmem" und "Freemem", falls Ihnen das etwas sagt (falls nicht: erstere fordert: "Bitte ein Kilo Byte!" vom Betriebssystem Speicher an, letztere gibt ihn wieder zurück). Somit schränken dynamische Variablen nicht Ihren Stack-Speicher ein. Es wird aber bei "New" quasi Buch geführt, welche Speicherbereiche reserviert wurden. Dadurch kann es zum einen bei "Dispose" nicht zu einem Absturz kommen (Guru Meditation Number $81000009), und zum anderen wird der Speicher am Programmende wieder ordentlich freigegeben

Und noch eine Besonderheit hat KICK-PASCAL im Zusammenhang mit Pointern auf Lager: Wenn Sie einer beliebigen Variablen ein "^" voranstellen, erhalten Sie einen Zeiger auf diese Variable.

Beispiel:

```
Program Pointer2;
 Var i:integer; p:^integer;
 Begin
  i:=26731; p:=^i; writeln(p^)
 End.
```

Ein anderes Beispiel: "MyScreen" sei der Zeiger, den Ihnen die Intuition-Funktion "OpenScreen" (Typ: ^Screen) zurückgegeben hat.

Mit der Prozedur "ClearScreen" aus der Graphics-library wollen Sie nun diesen Bildschirm löschen. Dazu brauchen Sie einen Zeiger auf den sog. Rastport des Screens, der als Feld (also nicht in Form eines Pointers!) in der Screenstruktur (Struktur=Record) enthalten ist. Der Befehl sieht so aus:

```
Clear Screen(^MyScreen^.Rastport);
```

Die Schreibweise mit den zwei "^" ist zugegebenermaßen etwas ungewohnt. Hier sehen Sie auch, daß die Auswertung quasi von rechts nach links erfolgt: Das erste "^" bezieht sich auf den ganzen Ausdruck "MyScreen^.Rastport" und nicht nur auf "MyScreen" (zumal ein Zeiger auf einen Zeiger keine allzu nützliche Sache ist).

Natürlich sollten Sie sich hüten, einen Pointer zu DISPOSEn, den Sie mit "^" auf eine NICHT durch New erzeugte Variable gesetzt haben. Die Folge wäre aber nur eine Fehlermeldung, denn KICK-PASCAL paßt hier auf.

3.7 SET-Typen

Sets können ziemlich groß werden: der Elementbereich darf bis zu 2^16 - 16 = 65520 Werte umfassen, wobei stets die untere Grenze zur nächsten durch 16 teibaren Zahl abgerundet und die obere zur nächsten höheren (!) durch 16 teilbaren Zahl aufgerundet wird. So ist z. B.

```
VAR s: SET of 0..65519;
```

eine der größtmöglichen Mengen im LongInt-Bereich. "1..65520" geht nicht, denn hier wird die 1 auf 0 ab- und die 65520 auf 65536 aufgerundet. Im Integerbereich wäre entsprechend

```
VAR t:set of -32752..32767;
```

gerade noch möglich - also fast schon "Set of integer"!

Elementtypen können alle geordneten Datentypen sein, also Char, Boolean, Ganzzahl- und Aufzählungstypen sowie Ausschnitte daraus.

Insgesamt gibt es folgende Verknüpfungen auf Mengen:

*	Schnittmenge:	[1..8]*[5..10] = [5..8]
+	Vereinigung:	[1,2] + [2,6..10] = [1,2,6..10]
-	Differenz:	[-10..10] - [-Maxint..0] = [1..10]
=	Mengenvergleich:	[17,4,21] = [4,17,21] ist TRUE.

<>	Gegenteil von =		
<=	Teilmenge:	[1,7] <= [0..10]	ist TRUE.
>=	genau andersrum:	[0..10] >= [1.7]	ist TRUE.
IN	"ist Element von":	5 IN [1..10]	ist TRUE.

Auch hier gelten die normalen "Punkt- vor Strichrechnung"-Regeln:

```
x IN a-b*c      entspricht      x IN (a-(b*c))
```

Sie sollten aber immer bedenken, daß Mengenoperationen in KICK-PASCAL (wie in wohl fast allen anderen PASCAL-Implementierungen) nicht gerade schnell sind. Genaugenommen sind sie sogar total lahm, vor allem dann, wenn die beteiligten Mengen groß werden. Deshalb sollten Mengen solchen Verwendungszwecken vorbehalten bleiben, bei denen es nicht auf Geschwindigkeit ankommt.

Eine Ausnahme gibt es aber: Falls hinter "IN" eine Mengenaufzählung steht, die ausschließlich aus Konstanten besteht, wird dies von KICK-PASCAL zu einer Folge von Vergleichsoperationen optimiert. So ist z. B.

```
L in [0,10..20,100..120]
```

erheblich schneller als

```
(L=0) or (L>=10)and(L<=20) or (L>=100)and(L<=120).
```

3.8 Strukturierte Datentypen: Arrays und Records

Strukturierte Variablen sind allgemein Zusammenfassungen mehrerer Variablen zu einer größeren Einheit. Es gibt in PASCAL zwei Arten von solchen Variablen: Feldtypen ("Arrays") und Verbundtypen ("Records").

Betrachten wir zuerst die Felder. Ein Feldtyp wird definiert durch

```
"ARRAY" "[" Indextyp "]" "OF" Elementtyp
```

Der Indextyp kann ein beliebiger geordneter Datentyp sein, also ein Ganzzahltyp, Char, Boolean, ein Aufzählungstyp oder ein Ausschnitt aus einem dieser Typen. In der Regel benutzt man hier aber Ausschnittstypen. Der Elementtyp kann ein beliebiger Datentyp sein.

Es wird jedem Wert des Indextyps eine Variable des Elementtyps zugeordnet.

Ein Beispiel:

```
VAR Feld: ARRAY[1..10] OF F Char;
```

Die Variable "Feld" besteht aus den zehn Elementen "Feld[1]" bis "Feld[10]", die jeweils Char-Variablen sind. Das Ganze ist übrigens identisch mit dem Datentypen "String[10]" (siehe Abschnitt 5 dieses Kapitels).

Alternativ hätte man auch so deklarieren können:

```
TYPE FeldIndex = 1..10;
     FeldTyp = ARRAY[FeldIndex] OF Char;
VAR  Feld: FeldTyp;
```

Der Elementtyp kann ein beliebiger Datentyp sein - also auch ein weiterer Array-Typ, z. B.

```
TYPE DreiD = ARRAY[-1..1] OF
             ARRAY['a'..'z'] OF
             ARRAY[Boolean] OF
             integer;
```

Das kann abgekürzt werden durch:

```
TYPE DreiD = ARRAY[-1..1, 'a'..'z', Boolean] OF integer;
```

Entsprechend kann man, wenn "Arr" eine Variable des Typs "DreiD" ist, ein Element wahlweise mit

```
Arr[ i+1 ] [ 'x' ] [ i=j ]
```

oder

```
Arr[ i+1, 'x', i=j ]
```

ansprechen.

Die Syntax für "Array-Typ" ist also:

```
"ARRAY" "[" GeordneterTyp { "," GeordneterTyp } "]" "OF"s Typ
```

Die Anwendungsmöglichkeiten von Arrays sind so vielfältig, daß jeder, der überhaupt schon einmal in irgendeiner Sprache programmiert hat, auf Anhieb zahlreiche Beispiele dafür aus dem Schatz seiner Erfahrungen nennen kann. Deshalb will ich Sie hier nicht mit einem Beispielprogramm dafür langweilen, sondern gleich zum zweiten strukturierten Datentyp übergehen: dem Record.

Ein Record- oder Verbundtyp faßt mehrere Variablen beliebigen, auch unterschiedlichen, Typs zu einer Verbundvariablen zusammen. Ein klassisches Beispiel aus der Mathematik: eine komplexe Zahl besteht aus einem Realteil und einem Imaginärteil, die jeweils reelle Zahlen sind. In PASCAL sähe das so aus:

```
TYPE Komplex = RECORD
                 Re: Real;
                 Im: Real
               END;
```

Das kann abgekürzt werden zu:

```
TYPE Komplex = RECORD
                 Re, Im: Real
               END;
```

Eine Variable "x" des Typs "Komplex" besteht dann aus zwei "Feldern", die "x.Re" und "x.Im" heißen und jeweils Real-Variablen sind.

Ein anderes Beispiel: Von 10000 Studenten sollen Vorname, Nachname (jeweils Strings mit bis zu 40 Zeichen) und Matrikelnummer (LongInt)) gespeichert werden. Dies läßt sich realisieren, indem man drei Arrays deklariert: Zwei Stringarrays für die Vor- bzw. Nachnamen und ein "ARRAY OF Long" für die Matrikelnummern. Aber sinngemäß gehören ja nicht alle Vornamen zusammen, sondern jeweils der Vorname, der Nachname und die Nummer jedes einzelnen Studenten. Um diese Zusammengehörigkeit im Programm klarzumachen, bietet sich die Verwendung von Records an:

```
VAR   Studies = ARRAY [1..10000] OF RECORD
                                      Vorname, Nachname: String[40];
                                      Matrikelnummer: Long
                                    END;
```

Eine Zuweisung an den k-ten Studenten könnte dann so aussehen:

```
Studies[k].Vorname:= 'Willi';
Studies[k].Nachname:= 'Wacker';
Studies[k].Matrikelnummer:= 3710815;
```

Einige Angaben zur Syntax:

```
Record-Typ:       "RECORD" Feldliste "END"
Feldliste:        { Bezeichner { "," Bezeichner } ":" Typ [ ";" ] }
                  [ Variantenteil ]
Variantenteil:    "CASE" [ Bezeichner ":" ] Typ-Bezeichner "OF"
                  { Konstante { "," Konstante } ":" "(" Feldliste ")"
                  [ ";" ] }
```

Bevor ich den "Variantenteil" erläutere, möchte ich darauf hinweisen, daß die Feld-Bezeichner einer jeden Feldliste einen eigenen Gültigkeitsbereich haben. Dies bedeutet, daß verscheidene Record-Typen gleichnamige Felder haben können und daß ein Feld einen Namen haben kann, der bereits vergeben ist. Einzige Bedingung ist, daß innerhalb eines einzelnen Records keine zwei Felder den gleichen Bezeichner tragen dürfen.

Oft kommt es vor, daß man in einem Recordtypen Daten speichern will, die sich gering unterscheiden. Beispiel: Es soll der Warenbestand eines Kaufhauses verwaltet werden. Wir begnügen uns zunächst mit der Warenbezeichnung, dem Verkaufspreis und einer Warengruppe. Bei Lebensmitteln wollen wir aber zusätzlich das Verfallsdatum und bei Kleidung die Konfektionsgröße speichern. Der erste Versuch mit einem normalen Record und einem Aufzählungstypen für die Kategorie könnte so aussehen:

```
TYPE  Warengruppe = (Lebensmittel, Kleidung, Sonstiges);

      Ware = RECORD
          Bezeichnung: STRING[50];
          Preis: RECORD    { Schachtelung von Record-Typen! }
                    DM: integer;   { Mark }
                    Pf: 0..99      { Pfennige }
                  END;
          Kategorie: Warengruppe;
          Verfall: RECORD
                     Tag, Monat, Jahr: integer
                   END;
          Konfgroesse: integer;
      END;
```

Der Haken an der Sache: Wir benutzen immer entweder das Verfallsdatum oder die Größe oder keins von beiden und verschwenden deshalb stets mindestens ein Feld. Das können wir mit einem Varianten-Teil ändern:

```
Ware = RECORD
         Bezeichnung: STRING[50];
         Preis: RECORD    { Schachtelung von Record-Typen! }
                  DM: integer;   { Mark }
                  Pf: 0..99      { Pfennige }
                END;
         CASE Kategorie: Warengruppe OF
           Lebensmittel: (Tag, Monat, Jahr: integer);
           Kleidung:     (Konfgroesse: LongInt);
           Sonstiges:    ( );
       END;
```

Die CASE-Zeile bewirkt zunächst einmal, daß ein Feld "Kategorie" des Typs "Warengruppe" eingerichtet wird. Der ganze Rest der Feldliste ist dann ein Varianten-Teil. Jede Variante beginnt mit einer Liste von Konstanten (im Beispiel bestehen die "Listen" jeweils nur aus einer Konstanten), gefolgt von einem Doppelpunkt - also ähnlich wie beim CASE-Statement. Dahinter folgt jeweils eine Feldliste - aber diesmal nicht mit "RECORD" ... "END", sondern von runden Klammern umschlossen.

Die Feldliste kann übrigens auch leer sein, wie Sie in der Zeile "Sonstiges" sehen. Was hat das dann zu bedeuten? Die Felder "Tag", "Monat" und "Jahr" belegen den selben Speicher wie das Feld "Konfgroesse". Die Zeile für den Fall "Sonstiges" ist vollkommen überflüssig, fördert aber die Lesbarkeit, weil dadurch klar wird, daß das Feld "Kategorie" auch diesen Wert annehmen können soll. Als Beispiel dafür, wie man mit solchen Records arbeitet, hier eine Prozedur zur Warenausgabe:

```
PROCEDURE Ausgabe(W: Ware);
  BEGIN
    writeln(W.Bezeichnung);
    writeln(W.Preis.DM , '.' , W.Preis.Pf);
    IF Kategorie=Lebensmittel THEN
      writeln('Haltbar bis: ', W.Tag, '.' , W.Monat, '.' , W.Jahr)
    ELSE
      IF Kategorie=Kleidung THEN writeln('Größe ', Konfgroesse )
  END;
```

Wie Sie sehen, sind die Felder des Variantenteils ganz normale Record-Felder, nur daß sie sich zum Teil überlappen. Eigentlich sollte es zu einer Laufzeit-Fehlermeldung führen, wenn man auf "Tag" zugreift, obwohl "Kategorie" den Wert "Kleidung" hat. KICK-PASCAL prüft dies aber nicht und überläßt es so Ihrer Verantwortung.
Der Variantenteil sieht zwar auf den ersten Blick wie ein CASE-Befehl aus, aber hier ist kein ELSE-Teil erlaubt. Auch können keine Ausschnitte wie "Konst1..Konst2" von Konstanten angegeben werden. Jeder Record kann höchstens einen Variantenteil haben. Falls Sie mehr benötigen, können Sie das über geschachtelte Records erreichen.

Beispiel: In unser Record soll noch die bevorratete Menge aufgenommem werden, entweder in Kilogramm, Litern (Real-Zahl) oder in diversen Stückzahlen (LongInt).

```
TYPE
  Warengruppe = (Lebensmittel, Kleidung, Sonstiges);
  Mass = (Masse, Volumen, Flaschen, Kisten, Stueck);

  Ware = RECORD
           Bezeichnung: STRING[50];
           Preis: RECORD
                    DM: integer;
                    Pf: 0..99
                  END;
           Menge: RECORD
                  CASE Einheit: Mass OF
                   Masse:                      (Kilogramm: Real);
                   Volumen:                    (Liter: Real);
                    Flaschen, Kisten, Stueck: (Anzahl: LongInt);
                  END;
           CASE Kategorie: Warengruppe OF
             Lebensmittel: (Tag, Monat, Jahr: integer);
             Kleidung:      Konfgroesse: LongInt);              {
Hier wurde die überflüssige Zeile weggelassen }
         END;
```

Der Mengen-Record besteht auschließlich aus einem Variantenteil (außer dem Feld "Einheit", das es immer gibt). In unsere Prozedur "Ausgabe" können wir nun an geeigneter Stelle folgendes einfügen:

```
  CASE W.Menge.Einheit OF
    Masse:    writeln(W.Menge.kilogramm , 'kg');
    Volumen:  writeln(W.Menge.Liter, 'Liter');
    Flaschen: writeln(W.Menge.Anzahl, 'Flaschen');
    Kisten:   writeln(W.Menge.Anzahl, 'Kisten');
    Stueck:   writeln(W.Menge.Anzahl, 'Stück');
  END;
```

Das Ganze hätte man mit einer WITH-Anweisung abkürzen können, aber das soll jetzt nicht unser Thema sein. Statt dessen noch einige Anmerkungen zu "Variant parts": das Feld hinter dem "CASE" " kann nicht nur - wie in unseren Beispielen - als Aufzählungstyp, sondern als beliebiger geordneter Typ definiert werden, d. h.

Boolean, ganzzahlig oder Char. In der Regel drängt sich die Verwendung eines Aufzählungstyps aber geradezu auf. Die Konstanten der Varianten müssen aber zu diesem Datentyp passen. Es ist nicht nötig, für alle denkbaren Werte eine Variante anzugeben, z. B. wurde im obigen Beispiel an Schluß die Zeile für "Sonstiges" weggelassen.

Es ist auch nicht erforderlich, ein Feld zum Abspeichern der Variante einzurichten. Wenn aus irgendeinem anderen Zusammenhang hervorgeht, welcher Zweig benutzt wird, reicht es, hinter dem "CASE" nur einen Typ-Bezeichner anzugeben.

Beispiel:

```
TYPE
  Komisch = RECORD
              Inhalt: STRING[10];
              Nummer: LongInt;
              CASE integer OF
                0, 1: (Flag: Boolean;
                       Operand1, Operand2: Real);
            2, 3, 4: (Parameter: Real);
                  5: (Fehlercode: integer;
                      CASE Boolean OF
                        true: (Re, Im: Real);
                        false: (Wert: Double)
                     )
               -1: ()
            END;
```

Hier haben wir auch ein Beispiel für Felddeskriptoren der Typen "Integer" und "Boolean" und für geschachtelte Varianten-Teile.

Eine Record-Variable mit Variantenteil belegt stets so viel Speicher, wie die längste Variante benötigt. Der Jensen-Wirth-Standard kennt für dynamische (also über Pointer referierte) Record-Variablen mit Variant Part eine besondere Version der "NEW"-Anweisung, bei der die gewünschte Variante angegeben wird. Wäre z. B. "P" ein Pointer auf den oben definierten Datentypen "Komisch", so würde

```
New (P, 5, false)
```

für die dynamische Variable "P^" genau so viel Speicher allozieren, wie die Variante "5" mit Unterrvariante "false" belegen würde,

```
New (P, -1)
```

würde entsprechend Platz für eine Variable mit leerem Variantenteil machen usw. Ich verwende bewußt den Konjunktiv, denn KICK-PASCAL unterstützt dieses Feature nicht. Sie vergeuden also evtl. etwas Speicherplatz.

3.9 Typkonvertierungen

KICK-PASCAL "managed" im allgemeinen die Kombination unterschiedlicher Datentypen selbstständig, so daß Sie sich darüber keine Gedanken zu machen brauchen. Der Compiler konvertiert Datentypen, wenn es sinnvoll ist (z.B. Integer ung LongInt) und meldet Fehler, wenn die Typen nicht zusammenpassen, z.B. Real und Pointer. Hin und wieder kommt es aber vor, daß Sie dem Compiler Ihren Willen aufzwingen müssen.

Beispiel:

```
Program IntToLong;
Var a,b:integer;
Begin a:=26731; b:=20000; writeln(a+2*b) End.
```

Da beide Variablen vom Typ "Integer" sind, rechnet das Programm in diesem Wertebereich - und es kommt zu einem Überlauf, denn der größte in "Integer" darstellbare Wert ist MaxInt=32767. Im Bereich "LongInt" könnte der Ausdruck aber korrekt ausgewertet werden.

Für solche (und andere) Zwecke gibt es in KICK-PASCAL die Möglichkeit zur Typumwandlung. Dazu verwendet man den Bezeichner des gewünschten Typs (hier: "LongInt" bzw. "Long") wie eine Funktion.

Beispiel:

```
... writeln(Long(a+2*b)) ...
```

Dies ist zwar syntaktisch richtig, bringt aber noch keine Verbesserung unseres kleinen Programms, denn der Ausdruck wird nach wie vor zunächst als "Integer" ausgewertet und erst danach in "Long" umgewandelt. Wenn Sie den Compiler dazu zwingen wollen, tatsächlich in "Long" zu rechnen, müssen Sie dafür sorgen, daß die Bestandteile des Ausdrucks vom Typ "Long" sind:

```
... writeln(Long(a)+Long(2)*Long(b)) ....
```

Dieser Ausdruck wird korrekt ausgewertet, ist aber etwas umständlicher als nötig. Es reicht ja schließlich, wenn ein Term "lang" ist, denn dann wird der Rest automatisch umgewandelt. Aber Vorsicht:

```
... writeln(Long(a)+2*b) ...
```

wäre falsch, denn (Punktrechnung vor Strichrechnung!) der Teil- ausdruck "2*b" wird zuerst in "Integer" ausgewertet und dann erst nach "Long" konvertiert! Richtig wäre dagegen:

```
writeln(a+Long(2)*b)
```
oder
```
writeln(a+2*Long(b)),
```

denn nun wird die Multiplikation korrekt als "LongInt" ausgeführt und dann die Addition ebenfalls, denn der zweite Summand ist ja lang.

Das klingt jetzt wahrscheinlich alles etwas verwirrend, aber in der Praxis werden Sie mit solchen speziellen Fällen wohl nur selten zu tun haben. Die Moral aus der Geschicht' ist aber, daß Sie im Zweifelsfall dem Compiler bei der Typkonvertierung nicht zuviel Freiheit lassen sollten.

Was wir hier mit "Long" gemacht haben, geht auch mit den vier anderen Ganzzahltypen. Sie können damit - falls das einmal sinnvoll sein sollte - Zahlen sogar "verkürzen", also bleistiftsweise von "Long" nach "Integer" oder von "Cardinal" nach "Byte".

Noch ein Beispiel:

```
Program Kardinal;
Var c:Cardinal;
Begin c:= 2*25000; writeln(c) End.
```

Falls Sie die Compiler-Option "Arithmetic Overflow" angewählt haben, wird ein Überlauf gemeldet, denn die hier auftretenden Zahlen 2 und 25000 liegen beide sowohl im Integer- als auch im Cardinalbereich.

In diesem Fall wählt der Compiler als Rechenbereich "Integer" (den Kontext, daß als Ergebnis eine Cardinalzahl zwecks Zuweisung an eine solche Variable gewünscht wird, beachtet er nicht). Auch hier können Sie ihn zum Rechnen in "Cardinal" zwingen, in dem Sie eine ("Cardinal(2)*25000" bzw. "2*Word(25000)") oder beide Zahlen ("Word(2)*Word(25000)") mit "Word" oder "Cardinal" zwangskonvertieren.

Bei all' diesen Beispielen ging es nur darum, einen korrekten Programmablauf zu gewährleisten. Es gibt aber auch die Möglichkeit, mit Typumwandlungen der Syntax ein Schnippchen zu schlagen, nämlich bei den Pointern. Denn auch Pointertypen und der Stringpointer "str" können untereinander umgewandelt werden. Außerdem können sie in LongInt-Zahlen und umgekehrt Zahlen in Zeiger verwandelt werden.

Beispiel:

```
... Var p:^Typ; ... New(p); writeln(Long(p)); ...
```

Dieses Fragment gibt die Adresse (als Zahl!) aus, auf die der Pointer "p" nach dem "New" zeigt.

Noch 'n Beispiel:

```
Program Speicherzugriff;
Var p:^Word;
    w:Word;
Begin
  p:=ptr($dff180);
  For w:=0 To MaxWord Do p^:=w
End.
```

Hier wird der Pointer "p" ganz gezielt auf die Adresse $dff180 gesetzt, wo nichts anderes als die Hintergrungfarbe des Bildschirms liegt. In der For-Schleife wird dann diese Farbe schnell geändert. Ergebnis: Ein Flimmern, an dem Ihr Augenarzt seine Freude haben würde.

Achtung: Setzen Sie einen Zeiger (fast) nie auf eine ungerade Adresse! Der Prozessor MC 68000 (steckt in Ihrem AMIGA drin - es sei denn, Sie besitzen ein 680x0-Turbo-Board mit x=2,3,4,5,.. oder einen Amiga 2500, 3000, 3500,...) kann 16- und 32-Bit-Zahlen nur von geraden Adressen lesen bzw. dahin schreiben. Bei ungeraden Adressen stürzt er ab (mit der Guru Meditation Number $00000003.xxxxxx). Dies sollten Sie also, wie gesagt, fast nie tun. "Fast" deshalb, weil 8-Bit-Daten (Byte, ShortInt, Boolean, Char) durchaus an ungeraden Adressen liegen dürfen.

Es gibt aber noch viele andere Möglichkeiten zur Typwandlung. Da wären zum einen die geordneten Datentypen, also Boolean, Char, Ganzzahl-, Aufzählungs- und Ausschnittstypen, die allesamt in beliebiger Kombination wandelbar sind. So können Sie z. B. "Char(n)" als Ersatz für die Funktion "Chr(n)" oder "Integer(x)" statt "Ord(x)" verwenden.

Das sollten Sie aber besser sein lassen, denn für solche Sachen gibt es ja die erwähnten Standard-PASCAL-Funktionen, während Typkonvertierungen eine Spezialität von KICK-PASCAL sind. Interessant ist allerdings die Möglichkeit, von Integer nach Aufzählungstypen zu konvertieren, also hier "ord" umzukehren:

```
Program Woche;
Type
Tage=(Montag,Dienstag,Mittwoch,Donnerstag,Freitag,Samstag,Sonntag);
Var  i: Integer;  t: Tage;
Begin
  For i:=0 to 6 Do
    Begin t:=Tage(i);
          If t In [Samstag,Sonntag] Then
            writeln('Wochenende!')
    End
End.
```

Desweiteren kann man alle geordneten Datentypen in Pointer und zurück wandeln. Auch in Fließkommatypen lassen sie sich wandeln ("Real('A')" würde z. B. die Real-Zahl 65 liefern), aber nicht umgekehrt.

Pointer kann man, wie erwähnt, in andere Pointertypen oder in alle geordneten Datentypen konvertieren. "Boolean(NIL)" wäre z. B. eine besonders umständliche Schreibweise für "false".

Die Datentypen "Real" und "Double" (bzw. "LongReal") lassen sich lediglich untereinander umwandeln, etwa wenn man eine bestimmte Rechengenauigkeit erzwingen will.

Hier ist eine Tabelle mit allen direkten Konvertierungsmöglichkeiten:

		nach:		
		geordnet	Pointer	Fließkomma
	geordnet	X	X	X
von:	Pointer	X	X	
	Fließkomma	•		X

(X = möglich, • = mit Umweg über "Trunc" und "Round")

3.10 Literale für Records und Arrays

Das AMIGA-Betriebssystem wurde in der Sprache C geschrieben. Dies hatte unter anderem zur Folge, daß geradezu exzessiv mit Structs (=Records) und über Zeiger verketteten Listen gearbeitet wurde.

Wenn Sie von PASCAL aus die Betriebssystemfunktionen nutzen wollen, müssen Sie deshalb immer und immer wieder Records initialisieren.

Ein Beispiel: Um mit der Intuition-Funktion "OpenWindow" ein Fenster zu öffnen, benötigt man eine Variable des Typs "NewWindow", der wie folgt definiert ist:

```
TYPE
  NewWindow = RECORD
   LeftEdge,TopEdge,Width,Height: integer;
   DetailPen,BlockPen: Byte;
   IDCMPFlags,Flags: Long;
   FirstGadget,Checkmark: ptr;
   Title: str;
   Screen,BitMap: ptr;
   MinWidth,MinHeight,MaxWidth,MaxHeight,_Type:integer
  END;
```

Jedem der Felder müssen Sie jetzt einen Wert zuweisen. In "normalem" PASCAL sähe das etwa so aus:

```
VAR Neu: NewWindow; ...
Neu.LeftEdge:=10; Neu.TopEdge:=20; Neu.Width:=620; ...
```

...oder so:

```
WITH Neu DO
  BEGIN
    LeftEdge:=10; TopEdge:=20; Width:=620; ...
  END;
```

Das ist natürlich sehr mühsam. Weil dies aber so oft nötig ist, bietet KICK-PASCAL Ihnen hier eine interessante Hilfe: Wertzuweisung auf einen Schlag! In unserem Beispiel sähe das so aus:

```
Neu:=NewWindow(10,20,6 0,160,$200,$100f,Nil,Nil,'Fenster',Nil,Nil,
               200,50,640,200,1);
```

Und dann ist die ganze Struktur schon initialisiert! An diesem Beispiel sehen Sie, wie die Syntax aussieht: Erst kommt der Typbezeichner und dann in runden Klammern, jeweils durch Komma getrennt, die Werte, welche die einzelnen Felder erhalten sollen. Falls Sie sich mit dem Amiga-Betriebssystem noch nicht so gut auskennen und Ihnen die Bedeutung dieses Records deshalb unklar ist: betrachten Sie es "'halt als irgendso 'n Record". Das vorher beschriebene geht natürlich mit jedem Recordtypen, also nicht nur mit denen, die zum Betriebssystem gehören, sondern auch mit den von Ihnen selbstdefinierten.

Welche Vor- und Nachteile hat das Ganze? Nun, es ist schön kurz. Andererseits müssen Sie die jeweilige Struktur genau kennen, denn hier kommt es auch auf die Reihenfolge der Felder an. Das ist aber nicht so schlimm: bei der "normalen" Record-Initialisierung müssen Sie die Struktur ebenfalls genau kennen, um zu vermeiden, daß Sie ein Feld vergessen. Der wohl gravierendste Nachteil ist, daß diese Syntax zu keinem anderen PASCAL kompatibel ist, schon gar nicht zu ISO-Standard-PASCAL. Ihre Programme sind dadurch nicht mehr auf andere Rechner oder Compiler übertragbar. Deshalb sollten Sie (im Rahmen der Aktion "sauberer programmieren") diese Art der Zuweisung nur im Zusammenhang mit AMIGA-Betriebssystem-Funktionen verwenden, die ja ohnehin nicht auf andere Rechnersysteme übertragbar sind. Hier ist es wohl auch am nötigsten.

Bei Records mit einem sog. Variant Part (mit "CASE") ist diese Art der Wertzuweisung nicht möglich. Dafür geht es aber bei Arrays ganz genau so:

```
PROGRAM ArrayDemo;
TYPE   Arr = ARRAY[1..10] of integer;
VAR
 Feld: Arr;
 i: integer;
BEGIN
 Feld =Arr(17,4,21,47,11,42,0,8,15,26731);
 FOR i:=1 TO 10 DO
   write(Feld[i]:7)
END.
```

Beachten Sie bitte, daß auch hier runde Klammern zu verwenden sind.

Mit einem "ARRAY ... OF CHAR" geht das übrigens nicht - ist aber auch gar nicht nötig, denn das ist ja ein String, und dafür gibt es bekanntlich viel einfachere Möglichkeiten der Wertzuweisung.

Seit Version 2.0 von KICK-PASCAL existiert auch die Möglichkeit, mehrdimensionalen Arrays und geschachtelten Records auf diese Weise einen Wert zuzuweisen. Dazu verwendet man einfach eine Klammerebene mehr.

Beispiel:

```
PROGRAM Array2Demo;
TYPE Arr2 = Array [ 1..3, 0..1 ] OF integer;
VAR Feld2: Arr2;
BEGIN
  Feld2:= Arr2((17,4), (42,0), (8,15)); );
END.
```

Denken Sie daran, daß dieses zweidimensionale Array lediglich eine Abkürzung für "ARRAY [1..3] OF ARRAY [0..1] OF integer" ist. So wird dann auch verständlich, daß wir hier drei Zahlenpaare haben und nicht etwa zwei Tripel. Bei höheren Dimensionsanzahlen geht das analog.

Analog behandeln wir auch geschachtelte Records:

```
PROGRAM RecRecRecDemo;
TYPE Rec = RECORD
             Name: RECORD
                     Vor,Nach: STRING
                   END;
             Adresse: RECORD
                        Strasse: RECORD
                                   Name: String;
                                   Hausnummer: integer
                                 END;
                        Ort: RECORD
                               Plz: integer;
                               Name: String
                             END
                      END
           END;
VAR Person: Rec;
BEGIN
  Person:= Rec(('Zaphod', 'Beeblebrox'),
               (('Industriestraße', 26), (6236, 'Eschborn')));
  WITH Person DO
    BEGIN
      writeln(Name.Vor, ' ', Name.Nach);
      writeln(Adresse.Strasse.Name, ' ', Adresse.Strasse.Hausnummer);
      writeln(Adresse.Ort.Plz, ' ', Adresse.Ort.Name)
    END
END.
```

Als kleines Extra funktioniert diese Schachtelung auch bei Records, die Arrays enthalten, und bei "Arrays of Records".
Ein Beispiel für letzteres:

```
PROGRAM ArrRecDemo;
TYPE KomplexVektor = ARRAY [1..5] OF RECORD
                                       Re, Im: Real
                                     END;
VAR x: KomplexVektor;
BEGIN
  x:= KomplexVektor((0,1), (-3,2), (42,0), (8,15), (17,4));
END.
```

3.11 Dateitypen

Eine Datei wird in PASCAL durch eine Variable repräsentiert, und die braucht auch einen Datentypen: einen FILE-Typen.

Die Syntax ist denkbar einfach:

```
Dateityp = "FILE" [ "OF" Typ ]
```

"Typ" gibt dabei den Typ der Daten an, die in der Datei gespeichert werden sollen. Dabei darf es sich nicht um FILE-Typen handeln - was sollte denn wohl auch "FILE OF FILE OF..." darstellen?

Es gibt einen vordefinierten FILE-Typ: "Text" ist "FILE OF Char". Und damit wären wir auch schon bei den beiden grundlegend verschiedenen Dateitypen: Es gibt Textdateien und Nicht-Textdateien. In Textdateien können Daten vieler unterschiedlicher Typen ein- und ausgegeben werden: Char, String, Boolean (nur Ausgabe) und alle Zahlentypen, und zwar bei Bedarf alle in dieselbe Datei. Bei der Ein- und Ausgabe werden die Daten als Folge von ASCII-Zeichen repräsentiert. Nicht-Textdateien, also solche, bei denen der Elementtyp nicht "Char" ist, sind an ihren festen Elementtypen gebunden. Dafür ist dieser aber beliebig (außer, wie bereits erwähnt, Dateitypen). In solche Dateien können also nur Daten eines ganz bestimmten Typs ausgegeben bzw. aus ihnen eingelesen werden. Die Daten werden in solchen Files im jeweiligen nicht unmittelbar lesbaren internen Datenformat abgespeichert.

Grob gesehen ist eine Datei eine beliebig lange Folge von Daten. Ein Schreib- oder Lesezeiger wandert von vorn nach hinten durch diese Datei. Auf das Dateielement, auf dem der Zeiger gerade steht, kann man durch die Puffervariable zugreifen: Ist eine Datei als

```
VAR Dat: FILE OF Typ;
```

deklariert, so heißt die Puffervariable dazu "Dat^" und ist vom Datentyp "Typ".

Zuerst muß man aber die Datei fürs Lesen oder Schreiben öffnen:

Reset(Dat)	öffnet die Datei, die bereits existieren muß, zum Lesen.
Rewrite(Dat)	öffnet sie für Schreibzugriffe. Falls sie noch nicht existiert, wird sie erzeugt.

Woher "weiß" das Programm aber, welche reale Datei, also welches File auf der Diskette o. ä., gemeint ist?

Dazu gibt es zwei Möglichkeiten:

- Man kann der Datei VOR dem Aufruf von Reset/Rewrite mit der Prozedur "Assign" einen String als Namen zuweisen. Beispiel:

```
Assign(Dat, 'DF1:Textdateien/Text1');
Reset(Dat);
```

Der Name wird dann nach dem "Assign"-Befehl in einem Puffer aufbewahrt. **Achtung:** Dazu stehen nur 30 Bytes zur Verfügung, so daß die Dateinamen nicht länger werden können!

- Man kann auch direkt beim Reset/Rewrite einen Dateinamen als zweiten Parameter angeben, z. B. so:

```
Reset(Dat, 'DF1:Textdateien/Text1');
```

Hier darf der Dateiname beliebig lang sein.

Falls die Datei nicht geöffnet werden konnte, hat die Funktion "IOResult" einen von 0 verschiedenen Wert, die Fehlernummer. Was nun geschieht, hängt davon ab, ob Sie "Reset" oder "Rewrite" verwendet haben: Im ersten Fall wurde die Datei zum Lesen geöffnet und enthält in der Regel Daten (es gibt natürlich auch leere Dateien), die Sie jetzt der Reihe nach lesen können, bis die Funktion "Eof(Dat)" wahr wird und so das Dateiende anzeigt. Dazu dient die Prozedur "Get":

```
Get(Dat)
```

liest das nächste Element aus der Datei und legt es in der Variablen "Dat^" ab.

```
Get(Dat); Variable:= Dat^;
```

kann ersetzt werden durch:

```
Read(Dat, Variable);
```

Wenn Sie eine Datei mit "Rewrite" zum Schreiben geöffnet haben, ist sie stets leer. Sie können nun Daten in die Datei schreiben, indem Sie sie zunächst in der Puffervariablen ablegen und dann mit der Prozedur "Put" schreiben.

Beispiel:

```
Dat^:= Ausdruck; Put(Dat);
```

Auch dafür gibt es eine Abkürzung:

```
Write(Dat, Ausdruck);
```

Soviel zur "normalen" Dateihandhabung von Standard-PASCAL. KICK-PASCAL erlaubt aber auch nicht-sequentiellen Dateizugriff, indem man mit der Prozedur "Seek" den Zeiger an eine beliebige Position setzen kann. Ferner kann man auf mit "Reset" geöffnete Dateien auch schreibend zugreifen, wobei jeweils die Daten, auf denen der Dateizeiger gerade steht, überschrieben werden.

Außerdem gibt es noch typfreie Dateien. Sie werden einfach als "FILE" ohne Typangabe deklariert, z. B. so:

```
VAR Datei: FILE;
```

In diesem Fall gibt es dann keine Puffervariable ("Datei^"), so daß auch die Prozeduren "Get" und "Put" nicht benutzt werden können. Nur mit den Prozeduren "BlockRead" und "BlockWrite" kann aus typfreien Dateien gelesen bzw. in sie geschrieben werden.

REFERENZ

KAPITEL IV

VARIABLEN UND KONSTANTEN

IV. Variablen und Konstanten

4.1 Standard-Variablen

KICK-PASCAL stellt folgende Standard-Variablen zur Verfügung:

Input, Output (Text)

Die Standard-Ein- und Ausgabedateien
Zeiger auf die Basisadresse der Dos-Library.

MathBase (Ptr)

Zeiger auf die "Mathffp.library"

MathtransBase (Ptr)

Dieser Pointer zeigt auf die Basisadresse der Mathtrans-Library. Aber Achtung: Diese Library liegt nicht im ROM, sondern muß von der Systemdiskette nachgeladen werden. Deshalb wird sie nur bei Bedarf geöffnet, d. h. sobald eine Funktion daraus aufgerufen wird.

FromWB (Boolean)

Jedes AMIGA-Programm kann im Prinzip auf zwei sehr unterschiedliche Arten gestartet werden: Von der Workbench oder vom CLI. Die Unterschiede liegen zum einen in der Art der Parameterübergabe (beim CLI-Start kann eine Zeichenfolge übergeben werden, beim Benchstart können evtl. andere Icons aktiviert sein) und zum anderen bei der Ein-/Ausgabe (im ersten Fall steht Ihnen dazu das CLI-Fenster zur Verfügung, im anderen Fall müssen Sie sich zuerst ein Fenster öffnen - bitte beachten Sie dazu auch das Demoprogramm "WB.p").

Die Variable "FromWB" gibt Ihnen nun die Möglichkeit, festzustellen, wie Ihr Programm gestartet wurde. Sie ist (klar) "true" genau dann, wenn der Aufruf von der Workbench erfolgte.

Die KICK-PASCAL-Entwicklungsumgebung simuliert übrigens einen CLI- Start, d. h. wenn Sie Ihr Programm von dort starten, hat FromWB stets den Wert "false".

ParameterStr (Str)

Dieser Stringzeiger zeigt auf die Zeichenfolge, die, wie oben erwähnt, beim CLI-Start dem Programm übergeben wurde.

ParameterLen (Integer)

Der Parameterstring endet nicht notwendig mit einem Nullbyte, deshalb enthält diese Variable die Länge.

StartupMessage (Ptr)

Beim Programmstart von der Workbench wird eine Message übergeben, die verschiedene Informationen enthält. Das Laufzeitsystem von KICK-PASCAL holt diese Nachricht ab und beantwortet (ReplyMsg) sie am Ende auch, so daß Sie sich darum nicht zu kümmern brauchen. Sie können sie aber mit Hilfe der Pointervariablen "StartupMessage" auswerten.

Mem (Array [0..MaxLongInt] of Byte absolute 0)

Dieses Feld repräsentiert den byteweise organisierten Hauptspeicher des Rechners. Dadurch sind einfache Speicherzugriffe möglich.

4.2 Vordefinierte Konstanten

Es gibt folgende vordefinierte Konstanten:

True	logischer Wert "wahr"
False	logischer Wert "falsch"
MaxInt	größte Integer-Zahl (32767)
MaxLong	größte LongInt-Zahl (2147483647)
MaxLongInt	anderer Name für "MaxLong"
MaxCard	höchste Cardinal-Zahl (65535)
MaxWord	anderer Name für "MaxCard"
MaxShortCard	größte 8-Bit-Cardinalzahl (255)
MaxByte	anderer Name für "MaxShortCard"
MaxShort	größte 8-Bit-Integerzahl (127)
MaxShortInt	anderer Name von "MaxShort"
Pi	= 3.14159265358979

REFERENZ

KAPITEL V

PROZEDUREN UND FUNKTIONEN

V. Prozeduren und Funktionen

KICK-PASCAL beherrscht alle Prozeduren und Funktionen des Standards - außer "Pack" und "Unpack", denn Datenstrukturen werden automatisch gepackt verwaltet. Daneben gibt es aber noch viele zusätzliche Befehle für alle Bereiche der PASCAL-Programmierung.

5.1 Ein- und Ausgabe

Write [(Write-Parameterliste)]

Syntax für Write-Parameterliste:

```
(Dateivariable | Write-Parameter) { "," Write-Parameter }
```

Syntax für Write-Parameter:

```
Ausdruck [ ":" Ausdruck [ ":" Ausdruck ] ]
```

Dies ist die Standard-Ausgabeprozedur. Sie kann prinzipiell ohne Parameterliste verwendet werden - dann gibt sie nichts aus und ist also überflüssig. Die Parameterliste ist eine geklammerte Liste von durch Komma getrennten "Write-Parametern" (siehe Syntax), von denen der erste auch eine Variable eines Filetyps sein kann. Falls letzteres nicht der Fall ist, so ist "Output" Default-Ausgabedatei.

Ist die Ausgabedatei vom Typ "Text", also "FILE OF Char", so haben Sie viele verschiedene Ausgabemöglichkeiten: die nachfolgenden Parameter dürfen von numerischen, booleschen, Char- und Stringtypen sein. In jedem Fall darf hinter dem Write-Parameter, mit einem ":" getrennt, die gewünschte Feldbreite für rechtsbündige Ausgabe stehen.

Fehlt sie, so gibt KICK-PASCAL grundsätzlich linksbündig aus. Bei der Ausgabe von Real-Werten darf man auch noch einen zweiten Parameter für das gewünschte Ausgabeformat angeben. Näheres dazu finden Sie im Abschnitt über Gleitpunkttypen.

Wenn die spezifizierte Ausgabedatei keine Textdatei ist, so müssen alle nachfolgenden Parameter zum Elementtypen der Datei typkompatibel sein. In eine Datei des Typs "FILE OF Real" könnte man z. B. einen beliebigen numerischen Wert schreiben. Wenn "f" eine Dateivariable eines solchen Typs ist, so wäre

Write (f, x, y)

äquivalent zu:

```
f^:= x ;  Put (f);
f^:= y ,  Put (f);
```

Näheres zu "Put" finden Sie im "Dateien"-Kapitel.

WriteLn [(Write-Parameterliste)]

Write-Parameterliste: Wie bei "Write"

Diese Prozedur ähnelt "Write", ist aber nur bei Textdateien möglich. Nach Ende der Ausgabe wird eine neue Zeile angefangen. Hier ist es desöfteren sinnvoll, die Parameterliste ganz wegzulassen: Dadurch wird ein Zeilenvorschub in der Standard-Ausgabe bewirkt.

Read [(Read-Parameterliste)]

Syntax für Read-Parameterliste:

```
Variable { "," Variable }
```

Die Standard-Eingabeprozedur. Wie bei "Write" kann auch hier die Parameterliste weggelassen werden, was aber keinen sinnvollen Befehl ergibt. Die Parameterliste, sofern vorhanden, ist eine Liste von durch Kommata getrennten Variablen. Analog zu "Write" darf auch hier der erste Parameter eine File-Variable sein, sonst wird "Input" als Default-Eingabekanal benutzt. Auch hier gibt es eine Unterscheidung zwischen Text- und Nicht-Text-Dateien:

Ist die Eingabedatei vom Typ "Text" (wie z. B. die Datei "Input"), so sind als Parameter Variablen von numerischen, Char- und Stringtypen erlaubt. Es werden dann jeweils Zeichen aus der Eingabedatei gelesen und in den gewünschten Typen konvertiert.

Bei Nicht-Text-Eingabedateien müssen alle Parameter Variablen sein, die zum Elementtyp der Datei typkompatibel sind, insbesondere dieselbe "Breite" haben müssen. Hier werden direkt die Daten und nicht eine lesbare Repräsentation der Daten gelesen.

Die Dateihandhabung von Standardpascal ist nicht mehr ganz zeitgemäß und mußte bei KICK-PASCAL leicht variiert werden:

```
Read(datei,variable)
```

entspricht deshalb:

```
Get(datei) ; Variable:= datei^
```

und nicht (Wirth: PASCAL Report, Kap. 11.4.1):

```
Variable:= datei^ ; Get(datei)
```

Des weiteren wird "EoF" erst "true", wenn das Dateiende wirklich erreicht wurde, d. h. eventuell (bei Textdateien) erst dann, wenn nichts mehr gelesen werden konnte und die Variable dadurch undefiniert ist. Dies läßt sich in der Regel durch Verwendung von "SeekEoF" statt "EoF" vermeiden.

ReadLn [(Read-Parameterliste)]

Wie Read, aber nur bei Textdateien möglich.

```
ReadLn(parameter)
```

entspricht

```
Read(Parameter); ReadLn
```

und "ReadLn" entspricht wiederum

```
While not eoln(datei) do Get(datei)
```

Im Klartext: Alles, was in der Eingabezeile noch hinter dem gelesenen Teil steht, wird überlesen.

Page[(datei)]

datei: Variable eines File-typs, Default ist "Output"

Erzeugt Seitenvorschub oder -wechsel. Beim Bildschirm bedeutet dies Löschen.

ClrScr [(datei)]

datei: Text-File

Anderer Name von "Page" (wurde aus Gründen der Kompatibilität implementiert).

GotoXY (x,y)

x, y: Integer

Setzt den Cursor in Spalte x, Zeile y. Die linke obere Ecke des Ausgabefensters hat die Koordinaten (1,1).

ClrEol

löscht die Zeile ab Cursor.

InsLine

fügt auf dem Bildschirm eine Zeile ein, d. h. der Text unterhalb der Zeile, in der der Cursor steht, wird nach unten gescrollt.

DelLine

Gegenteil von "InsLine": Die Zeile, in der der Cursor steht, wird gelöscht, alles, was darunter steht, um eine Zeile nach oben gescrollt.

EoF [(datei)]: Boolean

datei: Variable eines Filetyps, Default ist "input"

Dateiende- und Dateifehler-Funktion. EoF(input) ist übrigens immer "false".

Eoln [(datei)]: Boolean

datei: Variable des Typs "text", Default ist "input"

Meldet Zeilenende bei Read/ReadLn aus Textdateien.

SeekEoLN [(datei)]: Boolean

datei: Textfile-Variable, Default ist "input".

Diese Funktion entspricht dem normalen "EoLN", allerdings werden hier Leerzeichen überlesen. Beispiel:

```
Var i, j: integer;
Begin
  write('Bitte Zahlen eingeben: '); i:=0;
  Repeat
     read(j); i:=i+j
  Until EoLN;
  writeln('Summe: ',i)
End.
```

Dieses Programm liest von der Standardeingabe eine Zeile voller Zahlen und berechnet die Summe der Zahlen. Problem: Wenn der Benutzer hinter der letzten Zahl noch ein Leerzeichen eingibt, wird das Zeilenende nicht erkannt. Das kann behoben werden, indem man "EoLN" einfach durch "SeekEoLN" ersetzt.

SeekEoF [(datei)]: Boolean

datei: Textfile-Variable, Default ist "input".

Analog zu "SeekEoLN" prüft diese Funktion "EoF", wobei Leerzeichen und Zeilenenden überlesen werden. Im Gegensatz zum normalen EOF arbeitet "SeekEoF" nur auf Textfiles.

EmptyLN [(datei)]: Boolean

datei: Textfile-Variable, Default ist "input".

Diese Funktion liest eine Zeile aus der Textdatei in den Puffer, überliest führende Leerzeichen und stellt dann fest, ob die Zeile leer ist.

Anwendungsbeispiel: "Read" wartet beim Lesen von Zahlen stur darauf, daß eine Zahl eingegeben wird - da hilft kein RETURN-Drücken. Mit der EmptyLN-Funktion können Sie zuvor prüfen, ob der Benutzer etwas einzugeben gedenkt.

Beispiel:

```
Var i: integer;
Begin
  i:=$0815;
  writeln('Alter Wert: ',i);
  write('Neuen Wert eingeben oder mit RETURN bestätigen: ');
  If not EmptyLN Then readln(i);
  writeln('Neuer Wert ist: ',i)
End.
```

In dem Moment, wo die Funktion "EmptyLN" aufgerufen wird, wartet das Programm darauf, daß der Benutzer etwas eingibt und RETURN drückt, und es wird geprüft, ob die Eingabe leer ist, d. h. ob er einfach nur "RETURN" getippt hat. Die Funktion "EmptyLN" liest selbst aber noch nichts - abgesehen von Leerzeichen, die überlesen werden.

Nur wenn in der eingegebenen Zeile wirklich etwas steht, liest das obige Programm dies mit "Readln(i)" ein, andernfalls behält "i" den alten Wert.

5.2 Dateien

Reset (datei [, name])

datei: Variable eines FILE-Typs
name: Stringkonstante oder -Variable

Eine Datei wird zum Lesen und Schreiben geöffnet. Dabei kann - als Erweiterung des ISO-Standards - optional ein Dateiname angegeben werden. Wird er weggelassen, so muß zuvor mit dem "ASSIGN"-Befehl ein Dateiname zugewiesen worden sein.

Reset in der erweiterten Form (d. h. mit Dateinamen) übernimmt den String nicht in den Puffer, so daß einerseits keine Längenbeschränkung besteht, andererseits aber keine Stringausdrücke verwendet werden können.

Die Datei muß bereits existieren. Konnte sie nicht geöffnet werden, so wird kein Fehler gemeldet! Vielmehr gilt dann "Eof(datei)=true" und "IOResult<>0". Sie müssen den Fehler also selbst abfangen, bzw. das Laufzeitsystem meldet bei der nächsten Ein- oder Ausgabeoperation auf dieser Datei einen "File not open"-Error.

Rewrite (datei [, name])

datei: Variable eines FILE-typs
name: Stringkonstante oder -Variable

Eine Datei wird für Schreibzugriff geöffnet. Nach erfolgreichem "Rewrite" ist die Datei völlig leer, und der Lesezeiger steht am Anfang. Dies ist der (wichtige) Unterschied zu "Reset", wonach man außer lesen zwar auch schreiben kann, aber dabei lediglich den alten Dateiinhalt überschreibt.

Get(datei)

datei: Variable eines Dateityps

Daten in Puffer "datei^" lesen.

Put(datei)

datei: Variable eines File-Typs

Puffer "datei^" in Datei schreiben.

Close(datei)

datei: Variable eines File-Typs

Wenn Sie eine Datei mit "Reset" oder "Rewrite" geöffnet haben, müssen Sie sie nach Gebrauch mit dem "Close"-Befehl wieder schließen. Falls Sie das nicht tun, wird es am Programmende automatisch durchgeführt.

Assign (datei, name)

datei: Variable eines FILE-typs
name: Stringausdruck

Einer Datei wird ein externer Dateiname zugewiesen. Dies ist nötig, damit das Laufzeitsystem beim anschließenden "Reset" oder "Rewrite" dieser Datei weiß, welche reale Datei angesprochen werden soll.

Der Dateiname wird in einen zur Filevariablen gehörenden Puffer übernommen, der 30 Bytes lang ist. Mit dem Nullbyte am Ende kann ein mittels "Assign" zugewiesener Dateiname also maximal 29 Zeichen lang sein. Falls das nicht reicht, gibt es alternativ die erweiterten Versionen von "Reset" und "Rewrite" (s. u.), die ohne vorheriges "Assign" auskommen.

IOResult: integer

Nach jeder Ein- und Ausgabeoperation von KICK-PASCAL, also z. B. "Reset", "Rewrite", "Read", "Seek" usw. wird diese Funktion mit der vom Betriebsystem gelieferten Fehlernummer initialisiert, also "0", wenn alles klar ist, und einen Wert ">0", wenn ein Fehler auftrat.

Beispiel:

```
Var f: text;
Begin reset(f,'yxlegrümpf'); writeln(IOResult) End.
```

liefert, falls nicht zufällig eine Datei namens "yxlegrümpf" existiert, die Fehlernummer 205: "Object not found".

Eine Liste der Fehlernummern finden Sie in Ihrem AMIGA-Handbuch im Anhang B-5: "AmigaDOS Error Messages".

Filehandle (datei): LongInt

datei: Dateivariable

Diese Funktion liefert zu einer PASCAL-Dateivariablen die zugehörige Dateihandle, so daß Sie auf die Datei auch mit den Befehlen der Dos.library arbeiten können. Beispiel:

```
{$path "pascal:include/"; incl"libraries/dos.h" }

Var f: text;
    e: integer;
    c: Char;
Begin
  reset(f, 'CON:0/0/640/200/Kein CLI');  { Fenster öffnen }
  If IOResult=0 Then exit;               { Fehler abfangen! }
  e:= Execute('dir'$a'endcli',         { Befehlsfolge }
           filehandle(f),            { Eingabedatei }
           filehandle(f));           { Ausgabedatei }
  writeln(f);
  writeln(f,'Fehlercode war:',e);
  writeln(f,'RETURN drücken!');
  readln(f,c);
  close(f)
End.
```

Dieses Programm öffnet mit "Reset" ein Fenster und läßt darin vom DOS die Befehle "Dir" und "EndCLI" ausführen (letzterer ist wichtig, sonst hängt das DOS sich auf!). Die Befehle müssen durch das Zeichen #$a, also ein "Linefeed", getrennt werden.

Beachten Sie bitte: - "Filehandle" prüft nicht, ob die Datei auch wirklich geöffnet wurde. Falls dies nicht der Fall sein sollte, ist das Ergebnis undefiniert.

- Wenn das Programm aus dem PASCAL-System oder von der Workbench gestartet wird, sind "Filehandle(input)" und "Filehandle(output)" undefiniert. Wenn der Programmstart vom CLI erfolgte, liefern die beiden Ausdrücke die Handle der Standardein- bzw. Ausgabedatei.
- Mit DOS-Aufrufen können Sie die interne Dateihandhabung des PASCAL durcheinanderbringen. Generell sind "EoF", "EoLN", "FilePos" und "FileSize" einer Datei nach einem derartigen Aufruf undefiniert.
  "DosClose(Filehandle(...))" ist absolut tabu - Gurugefahr!

Seek (datei, position)

datei: File-Variable
position: Ganzzahliger Ausdruck

Setzt den Schreib-/Lesezeiger einer zuvor mit "Reset" geöffneten Datei auf eine bestimmte Position. Die erste Position hat die Nummer 0.

Eine "Position" ist ein Element des Datei-Grundtyps, bei "Text" also ein Char (=1 Byte), bei "File of integer" ein 2-Byte-Wort usw.

Nach "Seek(f)" ist die Puffervariable "f^" noch nicht initialisiert. Es muß also (wenn's unbedingt sein muß) noch "Get(f)" aufgerufen werden.

Filepos (datei): LongInt

datei: File-Variable

Gibt die Position des Schreib-/Lesezeigers in einer zuvor mit "Reset" geöffneten Datei an. Nach "Seek(datei,n)" liefert "Filepos(datei)" beispielsweise den Wert n, vorausgesetzt, die Datei ist entsprechend lang.

Filesize (datei): LongInt

datei: File-Variable

Liefert die Größe einer zuvor mit "Reset" oder "Rewrite" geöffneten Datei. Im allgemeinen wird die Größe aber nicht in Bytes oder Blöcken angegeben, sondern in Elementen des Datei-Grundtyps, so daß diese Funktion ohne Umformung zu "Seek" und "FilePos" kompatibel ist. Beispiel:

```
Var f: Text;
Begin
  reset (f, 's:startup-sequence');
  seek (f, filesize(f));
  writeln (f, 'echo "Hallo!"');
  close (f)
End.
```

Dieses Programm öffnet die Startup-sequence, setzt den Zeiger auf das Dateiende und hängt dort mittels "writeln" einen CLI-Befehl an.

Buffer (datei, groesse)

datei : Variable eines beliebigen FILE-Typs
groesse: gewünschte Puffergröße in Bytes

Die Dateizugriffe eines Programms erfolgen intern über die DOS-Library des Betriebssystems, die das Filesystem benutzt, welches wiederum über Devices (Gerätetreiber) auf die Hardware zugreift. Wir haben es also mit einem ziemlichen Aufwand zu tun, der jedesmal getrieben wird, bevor auch nur ein einziges Byte gelesen oder geschrieben werden kann.

Schlimmer noch: Dieser Überbau muß bei jeder einzelnen Dateioperation erledigt werden und ist vom Umfang der anschließend zu übertragenen Daten unabhängig. Sie können sich denken (oder mußten es bereits erfahren), was dann bei Textdateien, bei denen ja zeichenweise gelesen und geschrieben werden muß, passiert: Für jedes übertragene Byte muß die gesamte Maschinerie in Gang gesetzt werden. Selbst die wohl kaum als überragend schnell zu bezeichnenden AMIGA-Diskettenlaufwerke sind dann so schnell, daß die eigentlichen Diskettenzugriffe verglichen mit der

enormen Zeitverschwendung durch das Betriebssystem kaum noch Zeit brauchen. Vielleicht kennen Sie noch das IFF-Ladeprogramm, das bei KICKPASCAL 1.0 als Beispielprogramm beilag: Es war scheußlich langsam, aber nicht etwa, weil KICK-PASCAL so schlechten Code erzeugt hätte, sondern weil die Bilddaten im wesentlichen byteweise gelesen werden mußten.

Viel effektiver wäre es doch, für eine Datei einen großen Puffer im RAM einzurichten. Bei Schreiboperationen könnte man die Daten erst einmal in diesem Puffer ablegen und dann in einem Rutsch (und mit einem einzigen DOS-Aufruf) schreiben, sobald der Puffer voll ist.

Umgekehrt könnte man bei Leseoperationen zuerst einen großen Datenblock in den Puffer lesen und dann von dort ohne weitere DOS-Benutzung die Daten weiterverarbeiten.

Jaaa, werden Sie jetzt vielleicht sagen, das klingt zwar alles recht sinnvoll und einleuchtend, aber die Verwaltung eines solchen Pufferspeichers bedeutet doch einen hohen Programmieraufwand...? Keine Panik - KICK-PASCAL kann mit der Prozedur "Buffer" solche Speicher vollautomatisch (und für Ihr übriges Programm unsichtbar) verwalten. Die Prozedur benötigt nur zwei Parameter: zum einen eine Dateivariable, die zuvor mit "Reset" oder "Rewrite" geöffnet worden sein muß, und zum anderen die gewünschte Puffergröße in Bytes. Hier bringen wenige kBytes schon eine deutliche Geschwindigkeitserhöhung; über 10000 Bytes zu gehen, ist in der Regel nicht sinnvoll. Das Programm läuft dann ganz genau wie zuvor, nur daß eben die Dateioperationen beschleunigt werden.

Dazu noch ein paar Hinweise:

- Wurde die Datei mit "Reset" geöffnet, wird der Puffer nur für Leseoperationen benutzt und bei jedem "Write" geleert.
- Nach "Seek" wird der Lesepuffer automatisch entleert. Deshalb kann es passieren, daß ein Programm sogar verlangsamt wird, wenn es in einer Datei dauernd hin und her springt.
- Zwischen Öffnen und Schließen einer Datei kann die Puffergröße für diese Datei höchstens einmal festgelegt werden. Weitere Aufrufe von "Buffer" werden ignoriert.

BlockRead (datei, variable, anzahl)

datei: zum Lesen geöffnete Dateivariable
variable: beliebige Variable, in die Daten gelesen werden sollen
anzahl: Anzahl der zu lesenden Datenblöcke

Oft ist der Inhalt einer Datei nicht so strukturiert, daß man sie sinnvoll als "FILE OF <Typ>" deklarieren könnte. Man denke nur an die diversen IFF-Formate, die aus unterschiedlich aufgebauten Hunks verschiedenster Länge bestehen. Deshalb gibt es die Prozedur "BlockRead", die aus einer beliebigen Datei Daten beliebiger Art und Länge in eine beliebige Variable liest. "anzahl" ist dabei die Anzahl der

Datenblöcke, die jeweils dem Elementtyp der Datei entsprechen. Ist sie als "FILE OF Typ" deklariert, so werden anzahl * SizeOf(Typ) Bytes gelesen. Bei einer Textdatei entspräche ein Datenblock also einem einzelnen Zeichen, bei typfreien Dateien wird als Datensatztyp "Byte" angenommen.

Beispiel:

```
VAR t: Text;
    s: String[1000];
BEGIN
  Reset (t, 's:Startup-Sequence');
  If FileSize (t) >= 1000 Then
     Error ('Datei ist zu lang!');
  BlockRead (t, s, FileSize (t));
  s [FileSize (t) + 1]:= chr(0);    { Stringende markieren }
  Writeln(s)
END.
```

Das Programm liest die komplette Startup-Sequence in eine Stringvariable und gibt sie aus.

BlockWrite (datei, variable, anzahl)

datei: zum Lesen geöffnete Dateivariable
variable: beliebige Variable, in die Daten gelesen werden sollen
anzahl: Anzahl der zu lesenden Datenblöcke

Analog zu "BlockRead" schreibt "BlockWrite" Daten aus einer Variablen beliebigen Typs in eine Datei.

5.3 Arithmetische und andere Funktionen

Alle in diesem Abschnitt vorgestellten Funktionen sind Bestandteil des Jensen-Wirth-Standards. Einige von ihnen mußten natürlich für die zusätzlichen Datentypen von KICK-PASCAL erweitert werden. Sie sind in alphabetischer Reihenfolge aufgeführt.

Abs(num): Numerisch

num: Real- oder Ganzzahlausdruck

Betrag von Gleitpunkt- oder Ganzzahldaten. Das Ergebnis ist stets vom selben Typ wie der Parameter.

ArcTan(x): Real

x: Real-Ausdruck

Arcustangens-Funktion. Wie die meisten Real-Funktionen, wird auch diese der Mathtrans.library entnommen, welche bei Bedarf geöffnet wird. Die Library liegt aber nicht im ROM, sondern muß vom "libs:"- Verzeichnis der Systemdiskette geladen werden. Ist diese nicht vorhanden, wird ein Fehler gemeldet.

Chr(i): Char

i: Integer-Ausdruck, Bereich 0..255

Liefert das Zeichen mit dem Ascii-Code i. Umkehrfunktion ist "ord(c)".

Cos(x): Real

x: Real-Ausdruck

Die wohlbekannte Cosinus-Funktion. Bei dieser und allen anderen trigonometrischen Funktionen müssen die Winkel - wie in der Mathematik üblich - im Bogenmaß angegeben werden. Aus einem Gradwinkel erhält man die zugehörige Bogenmaßzahl, indem man durch 180 teilt und mit Pi multipliziert - die andere Richtung geht entsprechend genau andersrum.

Exp(x): Real

x: Real-Ausdruck

Dies ist die Exponentialfunktion zur Basis e (Euler'sche Zahl). Die zugehörige Umkehrfunktion ist "Ln".

Ln(x): Real

x: Real-Ausdruck, x>0

Ergibt den natürlichen Logarithmus von x.

Odd(i): Boolean

i: Ganzzahl-Ausdruck

Odd(i) ist "true" genau dann, wenn i ungerade ist.

Ord(o): Integer oder anderer Ganzzahltyp

o: Ausdruck eines geordneten Typs (Ganzzahl, Boolean, Char, Aufzählungstyp)

Liefert die Ordnungszahl des Ausdrucks in den Klammern. Ist dieser Ausdruck von einem Ganzzahltyp, ist das Ergebnis vom selben Typ und auch vom selben Wert (so daß die Funktion hier unsinnig ist). Andernfalls ist das Ergebnis vom Typ "Integer".

Pred(o): ordinal

o: Wie bei "Ord(o)"

Die Vorgänger-Funktion. Das Ergebnis ist stets vom selben Typ wie der Parameter und ist der Wert, dessen Ordnungszahl um 1 kleiner ist als der des Parameters. Beispiel:

```
Pred(26731)=26730, Pred('Y')='X', Pred(true)=false.
```

Pred(false) oder Pred(chr(0)) sind eigentlich undefiniert, es kommt aber zu keiner Fehlermeldung.

Round(x): LongInt

x: Real-Ausdruck, abs(x)<=MaxLongInt

Gerundeter Wert von x.

Sin(x): Real

x: Real-Ausdruck

Die beliebte Sinus-Funktion. Natürlich betrachtet auch sie den Parameter als Winkel im Bogenmaß (vgl. "Cos(x)").

Sqr(num): numerisch

num: Numerischer Ausdruck

Berechnet das Quadrat des Parameters. Das Ergebnis ist stets vom selben Typ wie der Parameter.

Sqrt(x): Real

x: Real-Ausdruck, x>=0

Berechnet die Quadratwurzel des Parameters

Succ(o): ordinal

o: wie bei "ord(o)" und "Pred(o)"

Diese Funktion ist das Gegenstück zu "Pred", d.h. sie liefert den Nachfolger eines Ausdrucks eines geordeneten Datentyps.

Trunc(x): LongInt

x: Real-Ausdruck, abs(x) <= MaxLongInt

Liefert den Ganzzahl-Anteil von "x". Im Gegensatz zu "Round" wird hier nicht gerundet; vielmehr werden die Nachkommastellen einfach weggelassen.

5.4 Pointer

New(point)

point: Variable eines Pointertyps

Speicherplatz für eine dynamische Variable reservieren und "point" darauf setzen. Bei Records mit Variantenteil wird immer die maximale Größe reserviert. Die Version "New(point,feld1,feld2,...)" ist nicht implementiert.

Dispose(point)

point: Variable eines Pointertyps

Dynamische Variable löschen, Speicher freigeben.

DisposeAll

Alle dynamischen Variablen und alle von Alloc_Mem reservierten Speicherbereiche werden freigegeben. Dies ist keine Standard-PASCAL- Prozedur.

5.5 AMIGA-Spezifisches

Alloc_Mem (len, cond): LongInt

len (LongInt): gewünschte Größe in Bytes
cond (LongInt): Bedingungen

Diese Funktion entspricht im Prinzip der Exec-Funktion "AllocMem".

Der AMIGA ist - wie Ihnen wohl kaum entgangen sein dürfte - multitaskingfähig, d.h. mehrere Programme können gleichzeitig laufen.

Das führt natürlich dazu, daß ein Programm nicht so ohne weiteres auf den Speicher zugreifen kann, denn es besteht ja immer die Gefahr, daß sich dabei zwei Programme ins Gehege kommen. Also muß sich ein Programm bei Bedarf vom Betriebssystem Speicher reservieren lassen. Genau das tut nun AllocMem bzw. Alloc_Mem. Dabei ist "Len" die gewünschte Größe des Speichers.

Bei "Cond" können Sie bestimmte Bedingungen angeben. Eine 2 (genannt MEMF_CHIP) bewirkt z. B., daß der Speicher im sog. Chipmemory liegt, bei $10000 (MEMF_CLEAR) wird der Speicher automatisch gelöscht (mit Nullen gefüllt). Wenn Sie keine besonderen Ansprüche stellen, geben Sie einfach cond=0 an.

Das Ergebnis der Funktion ist die Anfangsadresse des soeben reservierten Speichers. Was unterscheidet nun diese KICK-PASCAL-Funktion von der fast gleichlautenden Exec-Betriebssystem-Funktion?

Zum einen steigt Alloc_Mem mit einer Fehlermeldung aus, wenn kein Speicher reserviert werden konnte (z.B. weil schon alles belegt war). Zum anderen führt das Laufzeitsystem von KICK-PASCAL über die Alloc_Mem-Funktion quasi Buch, so daß alle reservierten Speicherbereiche am Programmende automatisch wieder freigegeben werden. Das können Sie aber auch "manuell" machen: Der entsprechende Befehl heißt Free_Mem.

Open_Window (X,Y,W,H,Farbe,IDCMP,Flags,Name, Screen, MinW,MinH,MaxW,MaxH): Ptr

X (Integer):	x-Koordinate der Fensterposition
Y (Integer):	y-Koordinate der Position
W (Cardinal):	Breite ("Width")
H (Cardinal):	Höhe ("Height")
Farbe (Cardinal):	Lo-Byte ist Hinter-, Hi-Byte Vordergrundfarbe
IDCMP (Long):	Signal-Bit-Maske
Flags (Long):	z.B. $1000 für ACTIVATE, 8 für WINDOWCLOSE
Name (Str):	Fenstertitel
Screen (Ptr):	Zeiger auf Screen, "Nil" für Workbenchscreen
MinW, MinH (Cardinal):	Mindestbreite und -höhe
MaxW, MaxH (Cardinal):	Maximalbreite und -höhe

Erraten: Open_Window öffnet ein Fenster. Wieder entsprechen die einzelnen Parameter den Feldern der NewWindow-Struktur, wie zuvor bricht das Laufzeitsystem ab, falls das Fenster nicht geöffnet werden konnte, und abermals werden die Fenster am Programmende automatisch geschlossen.

"FirstGadget", "Checkmark" und "BitMap" werden jeweils auf Nil gesetzt, "type" je nach Sachlage auf 1 oder 15.

Close_Window (p)

p: Ptr

Dies ist das Gegenteil von "Open_Window": ein Fenster wird geschlossen.

Free_Mem (Adr,Len)

Adr (LongInt): Speicheradresse Len (LongInt): Länge in Bytes

Dies ist das Gegenstück zu Alloc_Mem: ein reservierter Speicherbereich wird wieder freigegeben.

Open_Screen(X,Y,W,H,Depth,BackPen,DetailPen,Viewmodes, Name): Ptr

X (Integer):	x-Koordinate der Position
Y (Integer):	y-Koordinate der Position
W (Cardinal):	Breite ("Width") des Screens
H (Cardinal):	Höhe ("Height")
Depth (Integer):	Anzahl der Bitplanes ("Tiefe")
BackPen (Byte):	Hintergrundfarbe
DetailPen (Byte):	Vordergrundfarbe
ViewModes (Word):	Flags (z.B. 4 für Interlace, $8000 für Hires)
Name (str):	Titel

Diese Funktion eröffnet einen Screen. Die einzelnen Parameter sind identisch mit den entsprechenden Feldern in der NewScreen-Struktur.

Als "Type" wird stets 15 (Custom-screen) angenommen, "TextAttr", "Gadgets" und "CustomBitMap" werden auf Nil gesetzt. Konnte der Screen nicht geöffnet werden, bricht das Programm mit einer Fehlermeldung ab. KICK-PASCAL "merkt" sich, welche Screens mit dieser Funktion geöffnet wurden, und schließt sie ggf. am Programmende wieder automatisch.

Close_Screen (p)

p: Ptr

Dreimal dürfen Sie raten... Richtig! Screen schließen!

OpenConsole (p): Ptr

p (Ptr): Windowhandle

Diese Funktion eröffnet zu einem Window das Console.device. Zurück- gegeben wird eine Art Devicehandle. Dieser Pointer zeigt (abgesehen vom Offset 16 bzw. 64) auf den Read- bzw. Writereplyport.

ReadCon (p): Char

p (Ptr): ConsoleHandle (wird von "OpenConsole" zurückgegeben)

Liest vom Console.device ein Zeichen. Wurde kein Zeichen eingegeben, ist das Ergebnis chr(0). Beachten Sie aber bitte, daß viele Tasten des AMIGA-Keyboards eine ganze Folge von Zeichen liefern, die Sie dann nacheinander lesen müssen.

WriteCon (p,string)

p (ptr): Consolehandle string (str): Zeichenkette

Das Console-device "p" (erinnern Sie sich noch? Diesen Zeiger erhalten Sie von der OpenConsole-Funktion) wird beauftragt, eine Zeichenfolge im zugehörigen Window auszugeben.

CloseConsole (p)

p: Ptr

Diese Prozedur schließt ein Console-Device, das mit "OpenConsole" geöffnet worden war.

SetStdIO (p)

p: Ptr

Die Standard-Ein-/Ausgabe wird auf das Console-Device mit der Handle "p" umgeleitet. Dadurch können Sie auf ein mit "Open_Window" geöffnetes Fenster mit den normalen E/A-Prozeduren (Read, Write, ClrScr...) zugreifen.

SetStdIO(Nil)

Aktiviert wieder die normale Standard-E/A.

Im Handbuch zu KICK-PASCAL 1.0 wurde empfohlen, beim Programmstart von der Workbench mittels

```
Reset(Input, 'CON:...');   Output:= Input;
```

Kanäle für die Standard-E/A zu definieren. Dies ist bei Version 2.0 aus Kompatibilitätsgründen weiterhin möglich. Es ist aber "sauberer", obige Befehlsfolge durch

```
Win:= Open_Window (...);   Con:= OpenConsole (Win);   SetStdIO (Con)
```

zu ersetzen.

OpenLib(var,name,version)

var: Pointer-Variable
name (Str): Name der Library
version (LongInt): Versionsnummer

Diese Prozedur entspricht der Exec-Funktion "OpenLibrary". Die Unterschiede:

- Der Befehl ist nicht als Funktion, sondern als Prozedur realisiert. Die Basisadresse der Library wird dann mittels VAR•Parmeter zurückgegeben.
- Kann die Library nicht geöffnet werden, wird mit einer entsprechenden Fehlermeldung abgebrochen.
- Wie inzwischen schon fast gewohnt, wird die Bibliothek am Ende ordnungsgemäß wieder geschlossen.

CloseLib(p)

p (Ptr): Zeiger auf Library

Eigentlich ist dieser Befehl überflüssig, denn, wie oben erwähnt, werden Librarys automatisch geschlossen. Trotzdem wurde er der Vollständigkeit halber implementiert.

Wait_Port(p): Ptr

p (Ptr): Zeiger auf Message-Port

Entspricht der Exec-Funktion "WaitPort".

Wait(Mask): Long

Mask (LongInt): Signalmaske

Ist absolut identisch mit der gleichnamigen Exec-Funktion.

Get_Msg(p): Ptr

p (ptr): Zeiger auf Message-Port

Ist identisch mit der Exec-Funktion "GetMsg".

Reply_Msg(p)

p (Ptr): Zeiger auf Message

Identisch mit der Exec-Funktion "ReplyMsg".

5.6 Nützliche KICK-PASCAL-Spezialitäten

Break (mask): integer

mask: Bitmaske mit 4 Bits:

Bit	Wert	Bedeutung
0	1	Ctrl-C
1	2	Ctrl-D
2	4	Ctrl-E
3	8	Ctrl-F

KICK-PASCAL bietet Ihnen die Möglichkeit, Programme mit der Taste F10 abzubrechen. Das ist aber nicht sehr sauber: Das Laufzeitsystem fragt hier, um nicht mehr Zeit, als unbedingt nötig zu verlieren, direkt die Tastatur ab.Deshalb wird überhaupt nicht getestet, in welchem Fenster die Taste gedrückt wurde, so daß man damit eventuell mehrere Tasks gleichzeitig unterbricht. Aus diesem Grund ist diese Möglichkeit eher als Paniktaste während der Programmentwicklung gedacht - einem Anwender sollten Sie ein solches Programm nicht zumuten.

Langer Rede schwacher Sinn: Mit der Funktion "Break" können Sie die Tastenkombinationen ^C und ^D abfragen (und auch noch ^E und ^F - aber es ist nicht üblich, diese Tasten zu benutzen). Die Function "Break" gibt "true" zurück, wenn eine der in der Bitmaske enthaltenen Taste gedrückt wurde. An geeigneten Stellen Ihres Programms sollten Sie die ^C-Taste abfragen ("IF break(1) THEN...") und dann kontrolliert (geöffnete Dateien schließen, reservierten Speicher freigeben...) das Programm beenden. Spätestens wenn Sie eine lauffähige Objektdatei abspeichern, sollten Sie die "Unterbrechen"- Option des Compilers ausschalten.

CBreak

Testet, ob die Tastenkombination "Ctrl-C" gedrückt wurde, und bricht in diesem Fall mit der Meldung "BREAK" ab. Es entspricht damit der Anweisung " IF break(1) THEN error('BREAK') " und kann immer dann verwendet werden, wenn ein unkontrolliertes Abbrechen des Programms nicht schadet.

Nähere Erläuterungen zum Sinn dieses Befehls finden Sie bei der Beschreibung der Function "Break(n)".

FreeStack: LongInt

Gibt an, wie viele Bytes noch auf dem Stack frei sind. Man kann diese Funktion in rekursiven Programmen verwenden, um selbst Stacküberläufe abzufangen, sobald der Platz knapp wird.

Ein anderes Beispiel: Ein Programm mit großen Datenmengen.

```
Program Monstrum;

    Procedure Main;  { das eigentliche Hauptprogramm }
      Var a: Array[1..100000] of Real;
           { benötigt 4*100000 Bytes }
    Begin
           { hier steht das Programm }
    End;

Begin
  If FreeStack<410000 Then  { 400 KB für Array, 10 KB Reserve }
     Writeln('Zu wenig Stack, Babe!')
  Else
     Main
End.
```

Das Programm benötigt ein 400 KByte großes Array. Um eine Laufzeit- Fehlermeldung zu vermeiden, prüft das Hauptprogramm, ob genug Stack zur Verfügung steht. Nur falls dem so ist, wird die Prozedur "Main" aufgerufen, in der das Riesenarray lokal deklariert ist.

Random: Real

Liefert eine Zufallszahl mit 0 <= Random < 1. Der Zufallszahlengenerator benutzt ein Standardverfahren; zusätzlich wird aber auch die momentane Rasterposition des Bildschirm-Elekronenstrahls in die Rechnung aufgenommen, so daß die Folge richtig schön chaotisch wird.

Random (n): Integer

n: Integer

Liefert eine ganzzahlige Zufallszahl im Bereich von 0 bis n-1.

Randomize

Die Zufallszahlenfolge wird mit der momentanen Systemzeit initialisiert. Da in die Berechnung der Zufallszahlen sowieso ein nicht-mathematisches Element eingeht (s. o.), ist diese Prozedur bedeutungslos und wurde nur aus Kompatibilitätsgründen implementiert.

Hi (int): Integer

int: Integer

"Hi-Byte" einer Zahl: entspricht

```
Word(int) div 256
```

(da hier vorzeichenlos gerechnet wird), ist aber schneller.

Lo (int): Integer

int: Integer

"Lo-Byte": Niederwertiges Byte, genau wie "int AND $ff"

Upcase (c): Char

c: Char

Wandelt den Buchstaben "c", falls er zu den Kleinbuchstaben gehört, in einen Großbuchstaben. Natürlich werden auch Umlaute korrekt gewandelt. Beispiel:

```
Var s: String;
    i: integer;
Begin
  s:='O zerfrettelter Grunzwanzling!';
  For i:=1 to Length(s) do write(Upcase(s[i]))
End.
```

Ergebnis:

```
O ZERFRETTELTER GRUNZWANZLING!
```

Swap (int): Integer

int: Integer

Vertauscht Hi- und Lo-Byte des Parameters.

Frac (r): Real

r: Real-Zahl

Diese Funktion liefert die Nachkommastellen einer Real-Zahl.

Beispiel:

```
Frac(17.4) = 0.4
Frac(-0.07) = -0.07
Frac(1e+10) = 0    (außerhalb der Rechengenauigkeit)
```

Es gilt: x = Trunc(x) + Frac(x) , falls x eine Realzahl ist, die im "LongInt"-Bereich liegt, so daß "Trunc" den richtigen Wert liefert.

Addr (x): LongInt

x: Beliebige Variable, Prozedur oder Funktion

Gibt die Speicheradresse, an der die Variable "x" liegt bzw. die Prozedur oder Funktion "x" beginnt, zurück.

Die Funktion

```
Addr(var)
```

("var" = beliebige Variable) entspricht damit

```
Long(^var)
```

(siehe auch Kapitel "Pointer").

Pwr10(i): Real

i: Integer-Ausdruck, -19 < i < 19

Diese Funktion liefert die Zehnerpotenz einer ganzen Zahl als Real. Man könnte dies auch ohne weiteres durch wiederholtes Multiplizieren erreichen, aber "pwr10" holt sich die Werte aus einer Wertetabelle und ist damit erstens schneller und zweitens vermeidet man die Fortpflanzung von Rundungsfehlern.

DbPwr10(i): Double

i: Integer-Ausdruck, -308 < i < +308

Dies ist die Entsprechung zu "Pwr10" im Double-Bereich - daher auch das "Db" am Anfang.

SizeOf(x): Long

x: Typbezeichner oder beliebige Variable

Liefert den Speicherbedarf eines Datentyps bzw. einer Variablen:

```
SizeOf(Integer) = 2
SizeOf(DosBase) = 4    ("Dosbase" ist eine Variable des Typs
                        "Ptr", siehe auch Kapitel IV).
```

Delay(n)

n: Integerzahl

Delay macht - genau wie die gleichnamige DOS-Funktion - eine Pause von n 50stel Sekunden.

Error(string)

string: Str

Das Programm steigt mit der angegebenen Fehler- (oder sonstiger) Meldung aus.

Halt (nummer)

nummer: integer

Programmabbruch mit der angegebenen Fehlernummer.

AddExitServer (proc)

proc: globale Prozedur ohne Parameterliste

In einem Multitasking-System wie dem AMIGA ist es oft wichtig, daß Programme nicht unkontrolliert abbrechen, sondern zuerst noch irgendwie "aufräumen", z. B. Dateien schließen, Speicher freigeben usw. Hat man über KICK-PASCAL-Prozeduren auf derartige Resourcen zugegriffen (z. B. "Open_Window", "Reset", "Alloc_mem"...), werden die entsprechenden Schließ- oder Freigabeprozeduren am Programmende automatisch ausgeführt. Wenn man aber Betriebssystemfunktionen direkt aufgerufen, z. B. mit "AllocRaster" (aus der graphics.library) eine Bitplane reserviert hat, muß man selbst für ein ordentliches Programmende sorgen.

Natürlich kann man die entsprechenden Anweisungen einfach am Ende des Hauptprogramms aufrufen. Dann werden sie aber nur bei einem normalen Programmende ausgeführt, nicht beim Abbruch mit einer Fehlermeldung.

Nun stelle man sich folgendes Szenario vor: ein Programm reserviert sich mittels "AllocRaster" etliche große Bitplanes. Anschließend versucht es noch, mittels der KICK-PASCAL-Funktion "Open_Screen" einen Screen zu öffnen, beispielsweise in der Absicht, auf den (unsichtbaren) Bitplanes Bilder einer animierten Grafik zu berechnen und sie dann auf den (sichtbaren) Screen zu blittern. Nun sei das Chip-RAM durch die "AllocRaster"-Aufrufe aber schon so voll, daß kein Screen mehr geöffnet werden kann - das Programm steigt mit der Meldung "Intuition error" aus, ohne die Bitplanes wieder freizugeben. Ergebnis: das Chip-RAM ist fast voll belegt und kann nur mit einem Reset des Systems wieder freigegeben werden.

Hier hilft die Prozedur "AddExitServer": Mit ihr kann man Prozeduren angeben, die beim wie auch immer gearteten Ende eines Programms ausgeführt werden sollen. Durch wiederholten Aufruf von "AddExitServer" können beliebig viele solcher Exit-Prozeduren gewählt werden, die dann am Programmende in umgekehrter Reihenfolge ausgeführt werden. Die Prozeduren müssen global (also nicht lokal innerhalb einer anderen Prozedur oder Funktion definiert) sein und dürfen keine Parameter besitzen.

Beispiel:

```
PROGRAM Exitus;

 PROCEDURE Ausstieg;
    BEGIN
      Writeln('ExitServer ausgeführt.')
    END;
```

```
BEGIN
   AddExitServer(Ausstieg);
   Writeln('Division durch 0: ', 1/0);
   Writeln('Diese Zeile wird nie ausgegeben.')
END.
```

Das Programm erzeugt folgende Ausgabe:

```
Division durch 0:                <- "normale" Ausgabe des Programms
Division by zero.                <- Fehlermeldung des Laufzeitsystems
ExitServer ausgeführt.           <- Ausgabe des ExitServers
```

Bei jedem Aufruf der Funktionen "Open_Window", "Open_Screen" und "Open-Console" und "OpenLib" installiert das PASCAL-Laufzeitsystem im Prinzip nichts anderes als eine Exitserver-Prozedur, die die jeweilige Resource schließt. Wenn Sie also in Ihrem Programm zuerst einen Exitserver installieren und dann ein Window öffnen, wird am Ende des Programms erst (Aufruf der Exitserver in umgekehrter Reihenfolge!) das Fenster geschlossen und dann erst Ihre Exit-Prozedur aufgerufen. Sie können dann z. B. keine Ausgaben in das Fenster mehr machen.

Mit "New" oder "Alloc_Mem" reservierter Speicher wird dagegen erst nach der Abarbeitung aller Exitserver freigegeben (falls nicht schon geschehen), und noch geöffnete Dateien werden erst dann geschlossen.

Exit

Diese Prozedur verläßt den aktuellen Block. Wenn EXIT im Hauptprogramm aufgerufen wird, entspricht es einem GOTO-Sprung ans Programmende, in einer Prozedur oder Funktion entsprechend einem Sprung an deren Ende.

Beispiel:

```
PROGRAM Nanu;
   PROCEDURE p;
     BEGIN
       Writeln('P1');
       Exit;
       Writeln('P2')
     END;
 BEGIN
   Writeln('Start');
   p;
   Writeln('Mitte');
   Exit;
   Writeln('Ende.')
 END.
```

Das Programm erzeugt die Ausgabe:

```
Start
 P1
 Mitte
```

Exchange (var1, var2)

var1, *var2*: Variablen

Vertauscht den Inhalt zweier Variablen. Die Variablen müssen skalar sein, d.h. numerisch, Bool, Char, Aufzählungs- und Pointertypen.

Ferner müssen die Typen der Variablen kompatibel sein, insbesondere dieselbe Länge haben. Beispiel:

```
Var a, b: integer;
 Begin
   a:=26731; b:=4711;
   Exchange(a,b);
   write(a:8, b:8)
 End.
```

gibt aus:

```
    4711    26731
```

Inc (v)

v: Variable eines geordneten Typs (z. B. Ganzzahl, Char...)

Der Wert der Variablen "v" wird um 1 erhöht. Die Anweisung entspricht damit

```
v:= Succ(v),
```

Allerdings findet bei "Inc" keine Prüfung auf arithmetischen Überlauf oder Bereichsüberschreitung statt.

Beispiele:

```
{ Variablendeklaration: VAR c: Char; i: integer; b: Byte }

  c:= 'X'; Inc(c); Write(c)     gibt aus:        Y
  i:=  7 ; Inc(i); Write(i)     gibt aus:        8
  b:= 255; Inc(b); Write(b)     gibt aus:        0   (Überlauf!)
```

Dec (v)

v: Variable eines geordneten Typs (z. B. Ganzzahl, Char...)

Dies ist das genaue Geggenteil von "Inc": die Variable wird, ohne Bereichsunterschreitung oder Unterlauf abzufangen, um einen Wert erniedrigt.

5.7 Stringbehandlung

Concat(string1, string2, ..): String

Diese Funktion hängt beliebig viele Strings zusammen und gibt das Ergebnis als temporären String zurück. Diese Funktion ist im Prinzip überflüssig, da man Strings auch mit "+" aneinander hängen kann.

Copy(string, pos, len): String

String: Stringausdruck
pos: ganzzahlige Konstante
len: Länge (nicht-negativ)

Liefert den "len" Zeichen langen Ausschnitt aus "string" ab Position "pos".

Beispiel:

```
writeln(Copy('O zerfrettelter Grunzwanzling',6,7))
```

gibt "frettel" aus. Das Ergebnis ist übrigens stets ein temporärer String.
Falls "len" negativ ist, wird ein Fehler gemeldet.

Pos (string1, string2): integer

string1, *string2*: Stringvariablen oder -konstante (keine temporären Strings!)

Diese Funktion sucht, ob der erste String im zweiten vorkommt, und liefert die Anfangsposition.

Beispiele:

```
Pos('a' ,'Hallo')              = 2
Pos('mi' , 'Du mich auch!')    = 4
Pos('ha' , 'Hallo hallo')      = 7, denn Groß-/Kleinschreibung wird
                                    beachtet
Pos('' , 'Yxlegrümpf')         = 1
Pos('Himpel','Forever!')       = 0, denn "Himpel" kommt in "forever!"
                                    nicht vor.
```

StrLen(String): integer

String: Zeichenkette (String[n], Str, Stringkonstante, Char...)

Ermittelt die Länge des übergebenen Strings (also die Anzahl der Zeichen vor dem ersten Null-Char).

Length (String): integer

Ist identisch mit "StrLen".

Insert (string, stringvar, pos)

string: String-Ausdruck
stringvar: Variable eines STRING[n]-Typs
pos: ganzzahlige Position, pos >= 1

In eine Stringvariable wird an einer bestimmten Stelle ein Stringausdruck eingefügt. Falls die Position nicht positiv ist, außerhalb der bisherigen Stringlänge liegt oder falls das Ergebnis zu lang für die Stringvariable ist, wird ein Fehler gemeldet.

Beispiel:

```
VAR st, name: STRING;
BEGIN
   st:= 'Hallo!';
   Write ('Name: '); ReadLn (Name);
   Insert (' '+Name, st, 6);
   Writeln (st)
END.
```

Dieses Programm fügt den eingegebenen Namen und ein Leerzeichen vor dem "!" in die Variable "st" ein und gibt dann also "Hallo ...!" aus.

Delete (stringvar, pos, len)

stringvar: Variable eines STRING-Typs
pos: ganzzahlige Position, pos >= 1
len: ganzzahlige Länge, len >= 0

Aus einer Zeichenkette werden ab einer bestimmten Postion eine bestimmte Anzahl von Zeichen gelöscht.

Beispiel:

```
VAR st: STRING;
BEGIN
   st:= 'AMIGOXYA';
   Delete (st, 5, 3);
   WriteLn (st)
END.
```

In diesem Programm werden die Buchstaben "HI" aus dem String gelöscht.
Ergebnis:

```
AMIGA
```

RealStr(float, digits): String

float: Real-Ausdruck
digits: Stellenanzahl (ganzzahlig)

Wandelt die Real-Zahl in eine Zeichenkette und gibt diese zurück, und zwar exakt die Zeichenfolge, die mit "Write(float:0:digits)" ausgegeben würde. Näheres zur

Bedeutung des zweiten Parameters steht im Kapitel über die Numerischen Datentypen. Die Zeichenkette, die "RealStr" zurückgibt, ist temporär (siehe auch Kapitel "Stringtypen").

IntStr (i): tempString

i: LongInt-Ausdruck

Aus dem Ganzzahl-Ausdruck wird die entsprechende Dezimalzahl erzeugt und als temporärer String zurückgegeben.

Beispiel:

```
InsStr(17)      ergibt    '17'.
```

Val (string, numvar, intvar)

string: String-Ausdruck
numvar: Variable des Typs Real, Integer oder LongInt
intvar: Variable des Typs Integer oder Cardinal/Word

Diese Prozedur versucht, den Stringausdruck "string" in eine Zahl zu wandeln, und zwar abhängig vom Typ der Variablen "numvar" in eine Real-, Double-, Integer- oder LongInt-Zahl. Ist der String eine korrekte dezimale Zahldarstellung (ohne führende oder folgende Leerzeichen!), so erhält "numvar" den entsprechenden Zahlenwert und "intvar" den Wert 0. Falls der String nicht korrekt gewandelt werden konnte, ist "numvar" undefiniert und "intvar" enthält die Nummer des ersten Zeichens des Strings, das nicht paßt.

Beispiele:

```
Var R:Real; L:Long; I,p:Integer; ...

Val('4711', R, p)        ==> R=4.711E+4, p=0
Val('-5', L, p)          ==> L=-5,       p=0
Val('50000', I, p)       ==> I=undef.,   p=6  (weil 50000 > MaxInt)
Val('+3.14149', L, p)    ==> L=undef.,   p=3  (keine ganze Zahl)
```

REFERENZ

KAPITEL VI

COMPILER-ANWEISUNGEN, INCLUDEFILES UND BETRIEBSYSTEMFUNKTIONEN

VI. Compiler-Anweisungen, Includefiles und Betriebsystemfunktionen

6.1 Compiler Directives

6.1.1 Includefiles

Es gibt in PASCAL eine Klasse von Anweisungen, die nicht zum Programm gehören, sondern zur Steuerung des Compilers bestimmt sind. Sie sind leider nicht genormt. Es ist aber üblich, sie als Kommentare zu tarnen:

```
{$Anweisung}   oder   (*$Anweisung*)
```

Wie Sie sehen, sind diese Compiler Directives äußerlich nichts anderes als Kommentare, deren erstes Zeichen ein "$" ist. Der Compiler wird dadurch angewiesen, diesen Kommentar nicht etwa wie gewohnt zu überlesen, sondern die darin enthaltene Anweisung sofort auszuführen.

Aber gehen wir doch von der Theorie gleich zum ersten Beispiel:

```
{$incl "Dateiname" }
```

Diese Sequenz weist den Compiler an, eine Includedatei einzuladen.

"Was ist das?"

Es ist nicht immer sinnvoll, den gesamten Quelltext im Editor zu haben. Wenn Sie z.B. einige Prozeduren bereits fertiggestellt haben, können Sie diese in einer Textdatei ablegen und aus Ihrem Programm löschen, wodurch dieses kürzer und handlicher wird. Im Rest Ihres Programms weisen Sie den Compiler an der Stelle, wo vorher der ausgelagerte Teil gestanden hat, mit einer Directive an, diesen Teil als Includedatei zu laden. Er tut dann so, als würde der Inhalt der Datei an dieser Stelle im Quelltext stehen, und compiliert so, als wäre nichts geschehen.

Zur Programmierung des Betriebssystems stehen Ihnen einige fertige Hilfsdateien zur Verfügung, die Sie auf diese Weise in Ihr Programm übernehmen können. Doch dazu später mehr.

Alles, was hinter einer Directive steht, wird wie ein gewöhnlicher Kommentar überlesen:

```
{$incl"Hallo"  Diese Anweisung öffnet die Includedatei "Hallo" }
```

Sie können aber auch mehrere Directives hintereinander hängen, indem Sie sie mit einem Semikolon ";" trennen bzw. verbinden (kommt auf den Standpunkt an):

```
{$incl "Hallo";Incl"Horrido" ; include"Hallali" }
```

Wie Sie hier ebenfalls sehen, dürfen Sie statt "incl" auch "include" oder irgendetwas anderes schreiben - vorausgesetzt, es fängt mit "incl" an. Ferner zeigt obiges

Beispiel, daß die Groß- und Kleinschreibung bzw. das Einschieben von Leerzeichen dem Compiler egal sind.

Dieselbe Wirkung wie oben hätten Sie übrigens auch einfacher haben können:

```
{$incl "Hallo","Horrido","Hallali"}
```

Die Sache hat allerdings noch einen Haken: die Dateien werden aus dem aktuellen Verzeichnis geladen. Sie hätten natürlich auch einen ganzen Pfad angeben können, z. B. so:

```
{$incl "PASCAL:Includedateien/Gruß/Hallo" }
```

Wenn Sie aber mehrere solcher Dateien einladen wollen, wird es lang und umständlich. Zudem müßten Sie, wenn das Verzeichnis einmal wechselt (z. B. weil Sie auf der Diskette aufgeräumt oder sie umbenannt haben), alle Dateinamen ändern. Dagegen hilft eine andere Directive: PATH. Unser Beispiel sähe dann so aus:

```
{$path"PASCAL:Includedateien/Gruß/"; incl"Hallo","Horrido",Hallali"}
```

Dieses "path" gibt also einen String an, der in Zukunft vor alle Includedateinamen gehängt wird. Gegebenenfalls wird dabei ein "/" zwichen Pfad und Namen gesetzt. Seit Version 2.0 von KICK-PASCAL ist es auch möglich, einen Default-Wert für den Includepfad in der Config-Datei anzugeben. Diesen Pfad kann man auch mit dem Menüpunkt "Optionen/Suchpfade" ändern. Wenn man diesen Menüpunkt anwählt, erscheint ein kleiner Requester, wo man die Pfade in String-Gadgets eintragen kann und zwar je ein oder zwei Pfade für Includefiles und für Units. Trägt man nur im ersten Gadget einen Pfad ein, entspricht das der $Path-Directive, weiter unten erfahren Sie, wozu das zweite gut ist.

Übrigens: Sie können Includedateien auch verschachteln, d.h. eine Datei kann selbst wieder andere einladen.

Nun gibt es noch ein weiteres Problem: Der Compiler ist zu schnell! Er ist so unverschämt schnell, daß die Floppy beim Compilieren von Includefiles nicht mitkommt und der Compiler auf sie warten muß. Das ist natürlich Zeitverschwendung - zwar eine, mit der man leben kann, aber trotzdem ärgerlich. Deshalb gibt es bei der "Path"-Directive eine besondere Option: Sie können zwei verschiedene Pfade, durch ein Komma getrennt, angeben, z. B. so:

```
{$path "ram:include","pascal:include" }
```

Die Wirkung ist folgende: Bei "incl" wird die Datei zuerst im ersten Verzeichnis (hier: "ram:include/") gesucht. Ist sie dort vorhanden, wird sie auch von dort gelesen. Falls dem aber nicht so ist, holt der Compiler die Includedatei über den zweiten Pfad. Und jetzt kommt der raffinierte Trick: Sie wird nicht nur einfach gelesen, sondern gleichzeitig in das erste Verzeichnis kopiert, so daß sie beim nächsten Compilerlauf dort zur Verfügung steht! Auf diese Weise können Sie sich ganz einfach und automatisch die Dateien, die Sie benötigen, von der langsamen Diskette in die schnelle RAM- oder RAD-Disk kopieren...

Und es wird noch schöner: Beim Kopieren legt KICK-PASCAL bei Bedarf von selbst die Unterverzeichnisse an, sofern sie noch nicht existieren! Im obigen Fall würde also immer zuerst geprüft, ob das Directory "ram:include" bereits existiert. Zu diesem Thema noch ein Beispiel:

```
{$path "ram:include/","pascal:include/"}
{$incl "exec/tasks.h","exec/ports.h","devices/trackdisk.h"}
```

Hier werden drei der AMIGA-System-Includedateien eingeladen. Beim ersten Compilerlauf, wenn die Ramdisk noch leer ist, liest KICK-PASCAL die Dateien aus dem Verzeichnis "PASCAL:include", legt in der Ramdisk mehrere verschachtelte Unterverzeichnisse an (erst "include" und dann darin die Unter-Unter-Directories "exec" und "devices") und kopiert die vier Dateien dahin. Bei allen folgenden Compilierungen kann KICK-PASCAL dann wesentlich schneller auf die Includefiles zugreifen, so daß sie jeweils in Bruchteilen einer Sekunde übersetzt werden.

6.1.2 Bedingte Compilierung

Weitere Directives sind IF, ELSE und ENDIF. Sie erlauben Ihnen, den Compiler zu steuern. Hinter der IF-Directive ist eine Bedingung anzugeben, evtl. mit vorangestelltem "not". In der vorliegenden Version ist aber erst eine einzige derartige Bedingung implementiert: "DEF ident". Diese Bedingung ist wahr genau dann, wenn der dahinter angegebene Bezeichner definiert ist.

Ein Beispiel:

```
Program CompilerDirectives;
Const Deutsch=26731;  { Löschen, wenn Englisch erwünscht }
Begin
  {$if def deutsch}
  writeln('Hallo');
  {$else}
  writeln('Hello');
  {$endif}
  {$if def yxlegrümpf}
  Dieses hier wird immer überlesen.
  {$endif}
End.
```

Auf diese Weise können Sie Ihr Programm in zwei Sprachen gleichzeitig schreiben. Je nach dem, ob die Konstante (Sie könnten aber auch eine Variable, einen Typen, eine Prozedur... nehmen) "Deutsch" definiert ist, erfolgt eine deutsch- oder englischsprachige Ausgabe. Beachten Sie aber den gewaltigen Unterschied zur gewöhnlichen IF-Then-Else-PASCAL-Struktur: Dabei fällt die Entscheidung während der Laufzeit des Programms, hier schon während des Compilierens. Ferner wird hier das, was ausgeschlossen wird, vom Compiler völlig ignoriert, so daß es nicht einmal syntaktisch richtig sein muß.

Eine weitere Anwendung finden Sie in den meisten der mitgelieferten Includefiles. Unter diesen Dateien herrscht eine gewisse Hierarchie: einige benötigen selbst wieder andere Includedateien, um vom Compiler verstanden zu werden. Bevor diese

anderen Includes eingeladen werden, muß aber geprüft werden, ob der Compiler sie nicht vielleicht schon längst geladen hat, denn sonst würde es zu Fehlermeldungen wegen Doppeldeklarierungen kommen. Deshalb enthält z. B. die Datei "exec/devices.h" diese Zeilen:

```
{$if not def EXEC_DEVICES_H }
Const EXEC_DEVICES_H=0;
{$if not def EXEC_LIBRARIES_H; incl"exec/libraries.h";endif}
{$if not def EXEC_PORTS_H; incl"exec/ports.h";endif}
```

...Hier kommt der eigentliche Inhalt der Datei ...

```
{$endif}
```

Um die ganze Datei legt sich hier eine IF-ENDIF-Klammer, die sichert, daß diese Datei wirklich nur einmal vom Compiler übersetzt wird. Um dies feststellen zu können, wird innerhalb dieser Konstruktion eine Konstante namens "exec_devices_h" definiert. Ferner benutzt die Datei Bezeichner, die in den Dateien "exec/libraries.h" und "exec/ports.h" deklariert werden. Auch diese Dateien deklarieren Bezeichner, die wie die jeweilige Datei heißen. Damit kann der Compiler prüfen, ob er diese Dateien schon übersetzt hat, und sie nur dann öffnen, wenn dies nicht längst geschehen ist.

6.1.3 Error

KICK-PASCAL hat noch einige weitere Directives auf Lager, z. B. "Error". Damit können Sie sich individuelle Fehlermeldungen ausgeben lassen.

Eine denkbare Anwendung wäre z.B., wenn zwei Includedateien sich gegenseitig ausschließen, weil es zu Doppeldeklarierungen kommen würde. Ein Beispiel: Sie wollen ein Programm sowohl in einer deutsch- als auch in einer englischsprachigen Version schreiben. Dazu deklarieren Sie alle Texte in zwei verschiedenen Dateien als String-Konstanten. Die Datei "Deutsch.texte" könnte dann etwa so aussehen:

```
{$if def Englisch_texte;
  Error "Sie können nur eine Sprache verwenden";
  else }
 Const
  Deutsch_texte= 0;
  Gruß          = 'Guten Tag!';
  Abschied      = 'Auf Wiedersehen!';
{$endif}
```

"Englisch.texte" wäre dann genau anders 'rum:

```
{$if def Deutsch_texte;
  Error"..ie können nur eine Sprache verwenden";
  else }
 Const
  Englisch_texte= 0;
  Gruß          = 'Hello!';
  Abschied      = 'Bye!';
{$endif}
```

Bei zwei so einfachen Includedateien ist es natürlich kein Problem, auch ohne derartige Fehlermeldungen die Übersicht zu behalten. Wenn Sie aber einen ganzen Haufen ineinander verschachtelter Dateien anlegen, kann dies recht nützlich werden. Denn wenn in einer Includedatei ein Fehler auftritt, setzt der Compiler den Editor- Cursor nicht etwa auf die fehlerhafte Stelle (die ja gar nicht im Arbeitsspeicher, sondern in einer Datei liegt), sondern auf die zugehörige "incl"-Anweisung.

6.1.4 Linkersteuerung

Mit der Compiler-Anweisung

```
{$link "datei1", "datei2", ... , "dateix" }
```

wird der Linker angewiesen, die angegebenen Dateien zu laden und zum compilierten Programm hinzuzulinken. Die Dateien werden, wenn der Compiler auf diese Directive stößt, sofort geladen; gelinkt wird aber erst nach dem erfolgreichen Compile-Vorgang.

Die Directive darf, wie alle anderen auch, an jeder Stelle des Quelltextes stehen. Im Gegensatz zu "incl" gibt es zu "link" keine Pfad-Anweisung und auch keine Umkopier-Möglichkeit.

Die Directive "ulink" arbeitet wie "link", holt die Datei aber aus dem in der Config-Datei angegebenen Unit-Verzeichnis. Alles weitere zu dieser Directive finden Sie im Kapitel VII: "Units und Module".

6.1.5 Compiler-Optionen

Mit dem Pull-Down-Untermenü "Optionen/Compiler" können Sie einige Compiler-Optionen einstellen. Der Nachteil: Diese Einstellungen wirken nur global, gelten also für das gesamte Programm, und es geht nicht aus dem Quelltext hervor, welche Einstellung sinnvoll ist, so daß man immer, wenn man einen Quelltext in den Editor lädt, die Optionen von Hand einstellen muß.

Deshalb gibt es eine Compiler-Directive, mit der man diese Einstellungen vornehmen kann. Sie heißt

```
{$opt para1, para2, ... , paraX }
```

Syntax für einen einzelnen Parameter:

```
(("t"|"i"|"b"|"s"|"a") ("+"|"-"|"0")) | "q"
```

Die Anfangsbuchstaben der Parameter entsprechen den Tasten im Tastatur-Untermenü "Preferences" und wirken auch so:

{$opt q } schaltet alle Optionen aus ("Quick")

{$opt x+ } die Option "x" (hier ist natürlich "t", "i", "b", "s" oder "a" einzusetzen) wird eingeschaltet - entspricht "ON" im Tastaturmenü oder einem abgehakten Punkt im Pull-Down-Menü.

{$opt x- } schaltet die Option "x" aus ("x" wie oben zu ersetzen).

{$opt x0 } die entsprechende Option wird auf den Wert geschaltet, der im Menü eingestellt ist. Dies ist natürlich nur sinnvoll, wenn die Option vorher mit "x+" oder "x-" ein- oder ausgeschaltet wurde.

Beispiele:

```
{$opt q,s+}
```

schaltet alle Optionen außer dem Abfangen eines Stacküberlaufs aus.

```
s:= 0;
 {$opt b-,a-}
FOR i:=1 to 1000 do s:= s + a[i];
 {$opt b0,a0}
```

In diesem Programmfragment werden die 1000 Elemente eines Arrays summiert. Um dies zu beschleunigen, werden vor der Schleife die Unterbrechungsmöglichkeit (mit Taste "F10") und die Prüfung auf Überlauf ausgeschaltet. Nach der Schleife verwendet der Compiler wieder die Einstellung des Menüs.

Die Optionen gelten also von der Stelle an, wo die "opt"-Directive steht, bis sie durch eine folgende Directive wieder anders gesetzt werden. Am Anfang, also vor einer "opt"-Directive, verwendet der Compiler natürlich wie gewohnt die im Menü eingestellten Optionen.

Nun aber das Wichtigste: Die Bedeutung der einzelnen Optionen.

t - Subrange testen

Bei Wertzuweisungen an als Ausschnittstyp deklarierte Variablen ist zur Laufzeit der Wertebereich zu prüfen. Aber man benutzt Ausschnittsypen ja gerade, um solche Bereichsüberprüfungen durchführen zu lassen, so daß man diese - defaultmäßig aktivierte - Option wohl nur ausschalten wird, wenn man eine Exe-Datei eines ausgetesteten Programms erzeugen lassen will.

Bei Zuweisung konstanter Werte wird der Bereich schon zur Compile-Zeit geprüft, und zwar unabhängig von der "t"-Option. Das folgende Programm wird also immer eine Compiler-Fehlermeldung verursachen:

```
Var n: 1..100;
Begin n:= 101 End.
```

Dagegen findet bei folgendem Programm die Prüfung zur Laufzeit statt, und zwar nur dann, wenn die Option "t" aktiv ist:

```
Var n: 1..100;
Begin
  n:= 100;  { im erlaubten Bereich }
  n:= n+1;  { Bereichsüberschreitung, erst zur Laufzeit feststellbar }
End.
```

i - Indexbereich bei Array-Zugriff

Diese Option sorgt dafür, daß bei jedem Zugriff auf ein Array-Element getestet wird, ob der Index im richtigen Bereich liegt. Man sollte diese Option nur ausschalten, wenn das Programm wirklich durchgetestet und debuggt ist, denn die Folgen können sonst gleichermaßen fatal (GURU!) wie heimtückisch (Fehler schwer zu finden) sein: Zwar liefert ein Lesezugriff auf ein nicht-existentes Feldelement nur einen undefinierten Wert, aber eine derartige Wertzuweisung schreibt einen Wert irgendwo in den Speicher. Meist (jedenfalls bei geringen Abweichungen vom deklarierten Indexbereich) liegt an dieser Stelle eine andere Variable, die dann aus scheinbar unerklärlichen Gründen einen anderen Wert erhält - viel Spaß bei der Fehlersuche... Es kann aber auch passieren, daß Sie das System zerschießen und Ihr AMIGA wieder 'mal nach Indien pilgert. Also Vorsicht beim Ausschalten dieser Option!

Beispiel:

```
{$opt i-}
Var a: Array[1..10] of integer;
    k: integer;
Begin
  k:= 42;
  a[0]:= 26731;    { FALSCHER INDEX!!! }
  writeln(k)
End.
```

Dieses Programm gibt "26731" aus... Lustig, nicht?

b - Unterbrechen

Es ist ausgesprochen ärgerlich, wenn ein Programm während eines Probelaufs unaufhaltbar in einer Endlos-Schleife hängenbleibt. Zwar kann man dann den Quelltext problemlos mit dem "Rescue"-Programm retten, aber man muß neu booten, um das Programm zu stoppen. Deshalb gibt es eine Taste zum Abbrechen von PASCAL-Programmen - nämlich die Funktionstaste "F10". Wenn die Option "b" aktiv ist, baut der Compiler an geeigneten Stellen Tastaturabfragen in den Code ein, und zwar überall dort, wo eine Endlosschleife entstehen könnte:

- Am Anfang von WHILE- und REPEAT-Schleifen
- Bei Labels
- Innerhalb jeder FOR-Schleife (die zwar theoretisch nicht endlos, praktisch aber ganz schön lang sein kann)
- Am Anfang von Unterprogrammen (endlose Rekursion!)

Um nicht mehr Zeit als unbedingt nötig zu brauchen, fragt das Programm die Tastatur hardwaremäßig ab - auf einem Multitasking-System gewiß nicht das Gelbe vom Ei, denn stellen Sie sich einmal vor, Sie lassen Ihr mit KICK-PASCAL entwickeltes Apfelmännchen-Programm im Hintergrund rechnen, während Sie schon das nächste Programm entwickeln und dabei im KICK-PASCAL-Editor mit der Taste F10 ans Textende springen...

Sie sehen also, daß die Abbruchmöglichkeit mit F10 nur als Notausstieg während der Programmentwicklung taugt. Wenn Sie ein Programm schreiben, das der Benutzer abbrechen können soll, ist es wohl besser, an geeigneten Stellen (zum Beispiel in Schleifen - siehe oben!) mit dem "CBreak"-Befehl oder der Funktion "Break(1)" die Tastenkombination "^C" abzufragen.

Wenn Sie ein Programm geschrieben haben, das länger rechnen soll (ein paar Minuten, über Nacht oder so), ist es oft keine leichte Entscheidung, ob man die Breakpoints setzen soll. Wenn Sie sie rauslassen, riskieren Sie, Ihr Programm nur noch durch einen Reset abbrechen zu können. Der Laufzeitbedarf der Tastenüberprüfung ist unterschiedlich, macht sich aber vor allem bei engen Schleifen bemerkbar: Die Laufzeit einer leeren FOR-Schleife steigt dadurch um ca. 75%, wenn die Schleife aber nennenswerte Anweisungen enthält, ist der Laufzeitaufwand eher marginal.

Beispiel:

```
VAR L,x:LONGINT;
BEGIN
    {$opt -q,b01
    x:= 0;
    FOR L:=1 TO 1000000 DO x:= x+1 (L=100000)
END.
```

Bei dieser kleinen Schleife steigt die Laufzeit durch die Abbrechmöglichkeit nur noch um ca. 25%, und wenn der Schleifenkörper mehr Anweisungen enthält (dürfte die Regel sein), ist der Zeitaufwand für die Tastaturabfrage im Vergleich zu den restlichen "Kosten" eines einzelnen Schleifendurchlaufs lächerlich gering.

s - Stacküberlauf abfangen

Auf dem Stack (Stapelspeicher) werden zur Laufzeit die nicht als STATIC deklarierten Variablen abgelegt (lesen Sie dazu bitte auch den Abschnitt "dynamische und statische Variablen" im Kapitel "Module").

Die Option "s" sorgt nun dafür, daß dabei stets auch geprüft wird, ob noch genug Speicher frei ist - andernfalls nimmt das Programm sich den Speicher einfach, egal, ob 'was frei ist oder nicht.

Auch diese Compiler-Option benötigt etwas (aber nicht viel) Laufzeit. Vor allem (falls überhaupt) macht sich die Verzögerung bei rekursiven Prozeduren bemerkbar - aber gerade dort ist sie eigentlich unverzichtbar, denn der Stackbedarf von Rekursionen läßt sich nur schwer abschätzen und wird auch nicht bei "Linear Stack Requirement" berücksichtigt.

a - Überlauf bei arithmetischen Operationen melden

Bei arithmetischen Operationen wird jeweils auf Über- oder Unterlauf geprüft. Dies kostet vor allem bei Additionen und Subtraktionen von Integer-Zahlen (aller

Längen, versteht sich) Laufzeit. Bei dem folgenden Programm erhöht es beispielsweise die Rechenzeit um etwa 25 Prozent. Bei Multiplikationen oder sämtlichen REAL-Operationen ist der Effekt nicht so gravierend, da diese Operationen ohnehin ewig dauern (nach Computer-Maßstäben).

```
VAR i,j,k,l,m:integer;
BEGIN
   {$opt q,a0}
   l:= 0;
   m:= 0;
   FOR i:=1 TO 50 DO
     FOR j:= 1 TO 50 DO
       FOR k:= 1 TO 100 DO
         BEGIN
           l:= i+j-k
           m:= l+k
         END;
END.
```

Achtung: Bei LongReal-Rechnungen kommt es nicht zu einer Fehlermeldung, sondern zu einer Exception, und zwar unabhängig von dem "a"-Compilerswitch!

6.1.6 Kurzauswertung boolescher Ausdrücke

Werten Sie doch einmal den folgenden logischen Ausdruck aus:

```
((1 < 0) and (2 < 3))  or  ((1 < 2) or (1 = 2))
```

Falls Sie ein wenig Erfahrung mit boolescher Algebra haben, werden Sie gar nicht alle Terme des Ausdrucks betrachten, sondern etwa so vorgehen: Der linke Teil der AND-Verknüpfung ist FALSE, also ist das Ergebnis unabhängig von der rechten Seite FALSE. Bei der zweiten OR-Verknüpfung ist der erste Term TRUE, also auch das Ergebnis dieses OR. Das Gesamtergebnis des Ausdrucks ist also TRUE.

Sinnvollerweise kann KICK-PASCAL logische Operanden genauso auswerten: Ist der erste Operand von AND logisch falsch oder der erste von OR wahr, wird der zweite Operand gar nicht erst betrachtet, denn der hat auf das Ergebnis keinen Einfluß mehr.

KICK-PASCAL 1.0 wertete noch immer beide Operanden aus, bevor es das Ergebnis berechnete. Aus Gründen der Kompatibilität gibt es in Version 2.0 aber einen Umschalter, mit dem man eine Voll-Auswertung erzwingen kann. Er wurde in Form einer Compiler- Directive realisiert und heißt wahlweise

`{$bool full}` oder `{$bool all}`.

Die Einstellung gilt dann, wie gewohnt, von der Stelle an, wo die Directive steht, bis zu der Stelle, wo wieder umgeschaltet wird. Das geschieht mit

`{$bool min}` oder `{$bool fast}`,

ganz wie Sie wollen.

Jetzt stehen Sie möglicherweise wieder an der Stelle, wo Sie ausrufen: "Was soll das?", oder vielleicht auch: "Cui bono?" Denn da es ja klar ist, daß die Kurzauswertung Zeit spart, hat man nicht das Bedürfnis, in einen langsameren Modus zu schalten. Oder?

Natürlich hat es auf das Laufzeitverhalten eines Programms keinen Einfluß, ob ein Teilausdruck wie "2<3" ausgewertet wird oder nicht.

Allerdings können Sie in booleschen Ausdrücken auch Funktionen mit Seiteneffekt betrachten, d. h. selbstdefinierte Funktionen, in deren Anweisungsteil globale Variablen verändert werden (schrecklicher Programmierstil!) oder die Ein- oder Ausgaben machen. Ein Beispiel für letztere Unart:

```
Program
Die_vogonische_Poesie_ist_die_drittschlechteste_des_Universums;

Function Frage: Boolean;
    Var c: Char;
    Begin
      write('Wollen Sie das Gedicht wirklich lesen? '); readln(c);
      Frage:= (c In ['j', 'J'])
    End;

 Function Bestatigung: Boolean;
    Var c: Char;
    Begin
      write('Sind Sie ganz sicher? '); readln(c);
      Bestatigung:= (c In ['j', 'J'])
    End;

 {$bool full - bewirkt hier fehlerhaften Programmablauf! }

Begin
    If Frage and Bestatigung Then
      writeln('Na gut, Sie haben es nicht anders gewollt:'\10\10,
      'O zerfrettelter Grunzwanzling!'\10,
      'Dein Harngedrange ist fur mich'\10,
      'Wie Schnatterfleck auf Pieneuntich...'\e'0;33mWu',
      ''\e'Buu'\e'Bua'\e'Ba'\e'Be'\8\e'Bi'\8\e'Br'\8\e'Bg'\8,
            ''\e'B.'\8\e'B.'\8\e'B.')
End.
```

Jede der beiden Functions fragt den User, ob er sich das Gedicht wirklich antun will, und gibt die Antwort als booleschen Wert zurück.

Normalerweise steigt das Programm aus, wenn die erste Frage nicht mit "Ja" beantwortet wird. "Bool Full" bewirkt aber, daß dann trotzdem noch die zweite Frage gestellt und erst dann der Ausdruck ausgewertet wird. Natürlich kann man sich auch umgekehrt Programme konstruieren, die ausschließlich mit "Bool Full" korrekt laufen, aber das dürfte in der praktischen Programmierung kaum vorkommen, so daß defaultmäßig Kurzauswertung vorgenommen wird. Wenn Sie aber ein mit KICK-PASCAL entwickeltes Programm portieren wollen, sollten Sie daran denken, daß nicht jeder Compiler Kurzauswertung beherrscht - zum Beispiel die Version 1.0 von KICK-PASCAL!

6.1.7 Behandlung von Ein-/Ausgabefehlern

Die saubere Programmierung von Ein- und Ausgabeoperationen ist oft eine lästige Angelegenheit, denn hier muß das Programm mit einem Benutzer, einem Peripheriegerät oder dergl. kommunizieren, wobei es praktisch immer zu Fehlern kommen kann. Typische Beispiele sind Fehleingaben des Benutzers, der Versuch des Lesens einer nicht vorhandenen Datei usw. Wenn man ein Programm nicht nur selbst benutzen, sondern auch anderen Anwendern zugänglich machen will, muß man bei jeder nennenswerten EA-Operation dafür sorgen, daß eventuell auftretende Fehler nicht zu einem Fehlverhalten des Programms führen (OK, bei einfachen Bildschirmausgaben mit Write[Ln] dürften in etwa nie Fehler auftreten).

Der Normalfall ist, daß nach einer EA-Operation die Funktion "IOResult" eine Fehlernummer <> 0 liefert, so daß das Programm den Fehler erkennen und in geeigneter Weise darauf reagieren kann. Seit Version 2.0 gibt es aber die Möglichkeit, Code erzeugen zu lassen, der bei einem EA-Fehler mit einer Fehlermeldung aussteigt. Das ist zwar zugegebenermaßen nicht die benutzerfreundlichste Art der Fehlerbehandlung (man hat gerade 1000 Adressen in sein Dateiprogramm eingetipt und dann steigt das Ding beim Abspeichern aus...), aber wenigstens weiß der Benutzer dann, das etwas schiefgelaufen ist.

Das funktioniert, wie alles in diesem Kapitel 6.1, über eine Compiler Directive. Sie heißt wahlweise "i" (ist kompatibler) oder "io" (ist aussagekräftiger) und wird von einem "+" (Fehlerkontrolle einschalten) oder einem "-" (im folgenden Fehler ignorieren) gefolgt.

Beispiel:

```
VAR i, j: integer;
BEGIN
     Write ('Bitte zwei Zahlen eingeben: ');
     {$io+: ab jetzt bei Fehlern aussteigen }
     Read (i);
     {$io-: ab jetzt Fehler ignorieren }
     Read (j);
     IF IOResult <> 0 THEN
       Write ('Die zweite Eingabe war nicht richtig.');
     Writeln;
     Write ('Eingabe war: ', i:8, j:8)
END.
```

Bei Eingabe "x 4" steigt das Programm mit einer Laufzeitfehlermeldung aus, während es bei Eingabe "4 x" den Fehler erkennt und meldet.

Sie können sich die Funktionsweise der $io-Directive im Prinzip so vorstellen, als ob hinter jede Anweisung, die mit EA zu tun hat,

```
IF IOResult <> 0 THEN Error ('File error');
```

geschrieben würde.

Folgende Prozeduren werden von $io behandelt:

ClrScr
Get
Page
Put
Read
ReadLN
Reset
Rewrite
Seek
Write
WriteLN

Bei aktivierter $io-Option kann Ihr Programm noch nicht einmal Fehler, die beim Öffnen einer Datei auftreten, selbst abfangen. Deshalb ist es nicht sinnvoll, sie für das ganze Programm einzuschalten.

6.2 Libraries

Das Betriebssystem des AMIGA ist in mehrere Libraries aufgeteilt. Mit KICK-PASCAL können Sie alle diese Funktionen verwenden. Dazu müssen sie aber zuerst deklariert werden.

Jede Library besitzt eine Basisadresse. Unterhalb dieser Adresse liegt im Speicher stets eine Tabelle von Sprungbefehlen, die zu den jeweiligen Libraryfunktionen führen. Die Differenz zwischen der Basisadresse der Bibliothek und der Adresse des Einsprungs heißt der Offset der Funktion. Diese Offsets sind stets negativ (weil ja die Sprungtabelle, wie gesagt, unterhalb der Basisadresse liegt) und ist durch 6 teilbar (weil ein normaler Sprungbefehl in Maschinensprache 6 Bytes belegt).

Zunächst brauchen Sie also die Basisadresse der Library, die Sie benutzen wollen. Die Basen der "exec.library" und der "dos.library" stehen Ihnen durch die Variablen "SysBase" und "DosBase" zur Verfügung, bei allen anderen Bibliotheken erhalten Sie die jeweilige Basis mit Hilfe der Prozedur "Open_Library", die bereits im Kapitel "Prozeduren und Funktionen" erläutert wurde. Diese Adresse müssen Sie in irgendeiner Variablen ablegen, die vom Typ "Long"/"LongInt" oder einem Pointertypen sein kann.

Als nächstes sind die einzelnen Funktionen der Library zu definieren. Eine Library-Deklaration kann irgendwo im Deklarationsteil des Programms bzw. eines Unterprogramms stehen und hat folgende Syntax:

```
"Library" Basis ":" [Offset ":" ] {ProcOrFuncHeading [";"]} "End;"
```

Als Beispiel betrachten wir einen Teil der "exec.library", den Sie in dem Includefile "exec/lists.h" finden. Es beginnt mit dem Kopf:

```
Library SysBase:
```

Als Basis kann man einen beliebigen Ausdruck verwenden, aber eine einfache Variable ist wohl am sinnvollsten.

Nun folgen die einzelnen Prozeduren und Funktionen. Die erste heißt "Insert" und hat den Offset -234. Da es aber zu einer Kollision mit der gleichnamigen KICK-PASCAL-Prozedur kommen würde, mußte sie umbenannt werden, und zwar in "_Insert". Das ist möglich, denn die Betriebssystemfunktion "weiß" ja überhaupt nichts davon, unter welchem Namen sie aufgerufen wurde - Hauptsache, der Offset und die Parameter stimmen.

Also sieht die Deklaration so aus:

```
-234: Procedure _Insert(a0:p_List; a1,a2: p_Node);
```

Der Teil hinter dem ":" ist ein ganz normaler Prozedurkopf. Nun weiß der Compiler, daß dieser Procedure drei Parameter zu übergeben sind, von denen der erste vom Datentyp "p_List" und die beiden anderen vom Typ "p_Node" sind (diese beiden Datentypen wurden natürlich zuvor in dem Includefile definiert). Nun müssen die Parameter aber nicht in irgendwelchen Variablen, sondern in Prozessorregistern übergeben werden.

Der Prozessor 68000 hat die acht Datenregister d0, d1, .. d7 und acht Adressregister ("a0" bis "a7"). In allen diesen Registern können den Betriebssystemfunktionen Parameter übergeben werden - mit drei Ausnahmen: "a7" ist der Stackpointer, in "a6" wird unmittelbar vor dem Aufruf die Basisadresse gelegt und "a5" hat in KICK-PASCAL eine besondere interne Bedeutung. Diese drei Register dürfen also nicht verändert werden.

Zurück zum Beispiel: Es folgen noch sechs Prozeduren und eine Funktion:

```
-240:Procedure AddHead(a0:p_List; a1:p_Node);
-246:Procedure AddTail(a0:p_List; a1:p_Node);
-252:Procedure Remove(a1:p_Node);
-258:Procedure RemHead(a0:p_List);
-264:Procedure RemTail(a0:p_List);
-270:Procedure Enqueue(a0:p_List; a1:p_Node);
-276:Function FindName(a0:p_List; a1:Str):p_Node;
```

Man beendet die Library-Deklaration mit:

```
End;
```

Noch einmal das Ganze im Überblick:

```
Library SysBase:
 -234:Procedure _Insert(a0:p_List; a1,a2:p_Node);
 -240:Procedure AddHead(a0:p_List; a1:p_Node);
 -246:Procedure AddTail(a0:p_List; a1:p_Node);
 -252:Procedure Remove(a1:p_Node);
 -258:Procedure RemHead(a0:p_List);
 -264:Procedure RemTail(a0:p_List);
 -270:Procedure Enqueue(a0:p_List; a1:p_Node);
 -276:Function FindName(a0:p_List; a1:Str):p_Node;
End;
```

Aber keine Panik: Natürlich müssen Sie solche Tabellen nicht ständig selbst schreiben - warum sollten Sie das Rad immer wieder neu erfinden? Die Library-Deklarationen finden Sie in den Includefiles auf einer Ihrer KICK-PASCAL-Disketten. Genaueres dazu im nächsten Kapitel.

Alles, was Sie wissen müssen, ist, daß die Exec- und DOS-Library beim Programmstart immer schon geöffnet sind. Bei den anderen Libraries dürfen Sie nie vergessen, sie am Programmanfang mit "OpenLib" zu öffnen, denn andernfalls ist Ihnen der Guru sicher (ich habe das auch einmal vergessen und bin dann bei der Fehlersuche fast wahnsinnig geworden, weil ich nicht auf die Idee kam, mal am Programmanfang nachzusehen).

6.3 Anmerkungen zu den Include-Files

Die Include-Dateien enthalten alles, was Sie zur Programmierung des AMIGA-Betriebssystems benötigen: Konstanten, Typen (Strukturen) und Libraries. Sie entsprechen im wesentlichen den normalen "C"-Includefiles. Nur hin und wieder mußten die besonderen Gegebenheiten der Sprache PASCAL berücksichtigt werden.

Zuerst wären da Kollisionen mit PASCAL-Schlüsselwörtern zu erwähnen. Die beiden wichtigsten: einige Strukturen (z. B. "NewScreen") enthalten ein Feld mit dem Namen "Type", und in Exec gibt es einen Datentypen namens "Library". Diese Probleme wurden jeweils gelöst, indem diesen Bezeichnern ein Uniterstrich "_" vorangestellt wurde ("_Library", "_Type"). Genauso wurden die Problemfälle behandelt, die dadurch entstanden, daß PASCAL nicht zwischen Groß- und Kleinschreibung unterscheidet: in "intuition/intuition.h" gibt es sowohl die Konstante "CLOSEWINDOW" als auch die Prozedur "CloseWindow" sowie weitere derartige Kollisionen bei "reportmouse" und (in "intuition/screens.h") bei "showtitle". In derartigen Fällen hat stets eine Prozedur Vorrang vor einem Datentypen und dieser wiederum vor einer Konstanten, was in den genannten Fällen konkret heißt, daß den Konstantennamen stets ein "_" vorangestellt wurde.

Die dos.library enthält einige Funktionen und Prozeduren, deren Namen mit PASCAL-Bezeichnern kollidieren: "Read", "Write", "Close", "Seek", "Input" und "Output". Diese Funktionen erscheinen in den Includefiles jeweils zweimal: zum einen, wie gehabt, mit "_" am Anfang ("_Write"), zum anderen mit vorangestelltem "Dos" (z. B. "DosWrite"). Sie haben also die Wahl, welche Namen Sie benutzen wollen.

Weitere Kollisionen: Der Datentyp "Filehandle" in der Datei "libraries/dosextens.h" wurde in "_FileHandle" umbenannt, und die Funktion "Text" der graphics.library kann wahlweise als "_Text" oder als "GfxText" aufgerufen werden.

Bei KICK-PASCAL gibt es keine Datei "functions.h" wie in C, vielmehr sind die Funktionsdefinitionen auf verschiedene Includefiles verteilt: Die Funktionen der exec.library stehen in verschiedenen Dateien, und zwar jeweils da, wo die zugehörigen Datentypen definiert werden. Sie brauchen sich darüber also normalerweise

keine Gedanken zu machen. Für die anderen Libraries wurden entweder ".lib"-Files angelegt, in denen ihre Funktionen deklariert werden, oder sie wurden anderen Includefiles "beigepackt".

Hier ist eine Liste aller zur Verfügung stehenden Libraries, ihrer Basisvariablen und der Dateien, in denen sie deklariert werden:

NAME	BASIS	DATEI
clist.library	CListBase	clist.lib
console.library	ConsoleBase	devices/console.h
diskfont.library	DiskFontBase	libraries/diskfont.h
dos.library	DosBase	libraries/dos.h
exec.library	SysBase	exec/#?.h
expansion.library	ExpansionBase	libraries/expansionbase.h
graphics.library	GfxBase	graphics.lib
icon.library	IconBase	workbench/icon.h
intuition.library	IntBase	intuition.lib
layers.library	LayersBase	layers.lib
potgo.library	PotgoBase	resources/potgo.h
timer.library	TimerBase	devices/timer.h
translator.library	TranslatorBase	libraries/translator.h

Noch eine Bemerkung zu den zahlreichen Datentypen, die in den Includefiles definiert werden: Zu jedem Record-Typen existiert ein Zeigertyp, der den Namen des Record-Typs mit vorangestelltem "p_" trägt, z. B. "Window" und "p_Window". Der Grund dafür ist, daß an einigen Stellen (Parameterlisten) in PASCAL nur Typbezeichner wie z. B. "p_Window", aber keine Typbeschreibungen (wie "^Window") erlaubt sind.

REFERENZ

KAPITEL VII

UNITS UND MODULE

VII. Units und Module

7.1 Modulare Programmierung: was, warum und wie?

KICK-PASCAL bietet Ihnen seit Version 2.0 ein Feature, das (wie viele andere nützliche Dinge auch) in Standard-PASCAL fehlt: das Konzept der modularen Programmierung. Die Grundidee ist die, daß ein Programm in beliebig viele sog. Module aufgeteilt wird, die dann mehr oder weniger unabhängig voneinander ediert und compiliert und erst danach zu einem einzigen Programm zusammengebunden werden. Das Zusammensetzen der Module heißt "linken". Sicher ist Ihnen bereits die Ausgabe "Linking..." am Ende des Compilevorgangs von KICK-PASCAL aufgefallen.

KICK-PASCAL ruft den integrierten Linker immer automatisch auf, auch dann, wenn es bei einem nicht-modularen Programm scheinbar nichts zu linken gibt (gibt es aber doch: das Programm wird mit dem PASCAL- Laufzeitsystem zusammengebunden).

Es gibt viele gute Gründe, sein Programm in Module zu zerstückeln:

- Die einzelnen Quelltexte werden kürzer. Da man immer nur das Modul neu compilieren lassen muß, das man gerade bearbeitet, werden auch die Compilezeiten kürzer (bei KICK-PASCAL sowieso kein Thema).
  Auch der Workspace-Speicherbedarf sinkt (bei KICK-PASCAL sehr wohl interessant, denn das Programm benötigt ja einen zusammenhängenden Speicherblock, und da wird es schnell eng).
- Lange Programme werden überschaubarer, wenn man sie sinnvoll in Module aufteilt. "Sinnvoll" heißt hier, daß die Einteilung nicht willkürlich erfolgt, sondern die Module jeweils einen zusammengehörigen Bereich des Programms umfassen. Zum Beispiel könnte man eine Dateiverwaltung in die Module "Benutzeroberfläche", "File-Handhabung" und "Such- und Sortierprozeduren" unterteilen, was sich dann noch beliebig weiter verfeinern und zerkleinern ließe.
- Man kann sich aus immer wieder benötigten Programmteilen eine Bibliothek aus Modulen, die man bei Bedarf nur noch einlinken muß, zusammenstellen. Dafür eignet sich besonders das UNIT-Konzept.
- Viele Leute glauben auch, Module seinen für Programmentwicklungen im Team hilfreich, da dann jeder Programmierer seinen Teil des Projekts unabhängig von den anderen erstellen kann. Nun ja, T.e.a.m. steht bekanntlich für "Toll, ein anderer macht's", und der eine, an dem normalerweise letztendlich die ganze Arbeit hängen bleibt, kann auch ohne Modulkonzept auskommen. Sollten in einem Team jedoch wirklich einmal mehrere Leute arbeiten, so ist Unabhängigkeit alles andere als wünschenswert, denn es bricht regelmäßig ein Chaos

herein, sobald die Module zusammengefügt werden sollen (An genau dieser Stelle möchte ich mit einem herzlichen "Core Dumped!" das Juhunix-Propra-Power-Team grüßen). Wenn aber tatsächlich einmal der unwahrscheinliche Fall eintreten sollte, daß ein Team sowohl motiviert als auch koordiniert arbeitet, könnte sich das modulare Programmieren theoretisch als nutzbringend erweisen.

KICK-PASCAL bietet sogar zwei verschiedene Modulkonzepte, die in gewissen Grenzen miteinander kombiniert werden können. Sie sind gekennzeichnet durch die Bezeichnung UNIT oder MODULE. Zunächst wollen wir uns hier den Modulenn zuwenden, die für den Programmierer zwar etwas schwieriger zu handhaben, aber in gewissem Sinne flexibler einsetzbar sind.

Ein KICK-PASCAL-Programm besteht, da wir vorerst davon ausgehen, daß wir keine Units verwenden, immer aus einem Hauptprogramm und beliebig vielen Modulen. Jeder Teil ist nach dem Compilieren als sog. Objektdatei abzuspeichern, damit der Linker ihn in die anderen Teile einbinden kann. Das Hauptprogramm sieht wie gewohnt aus, während die Module mit "Module" statt "Program" beginnen und keinen Anweisungsteil auf oberster Ebene haben. Die Verbindung der Komponenten geschieht über gemeinsame Variablen und Procedures, die mittels der Directives "IMPORT" und "EXPORT" ausgetauscht werden.
Aber nun eines nach dem anderen:

7.2 Syntax von Modulen

Betrachten wir als erstes ein kleines Beispiel, an dem das Prinzip des Linkens schon klar wird. Hier ist ein Hauptprogramm:

```
Program Haupt;

Var i:integer; EXPORT;

Procedure p(a:integer); IMPORT;
Procedure q; IMPORT;

Function f(n:integer):integer; EXPORT;
   Begin
     i:=n*n
   End;

Begin { Hauptprogramm }
   p(10);
   q
End.
```

Dieses Hauptprogramm importiert zwei Prozeduren, nämlich "p" und "q", und exportiert die Funktion "f" und die Variable "i". Nun brauchen wir wenigstens noch ein Modul, das die importierten Procedures zur Verfügung stellt (es macht nichts, wenn die exportierten Bezeichner nirgendwo benutzt werden). Hier ist ein solches Modul:

```
Module Unterprogramme;

Var i: integer; IMPORT;

Procedure p(k:integer); EXPORT;
   Begin
     i:= round(sqrt(k))
   End;

Procedure q; EXPORT;
   Function f(j:integer):integer; IMPORT;  { lokale Funktion! }
   Begin  { von "q" }
     writeln('Funktionswert von ', i, ' ist ', f(i))
   End;

Begin { leerer Anweisungsteil }
End;
```

Zuerst zwei Anmerkungen zur Syntax: Daß die Direktieven "Import" und "Export" hier groß geschrieben wurden, hat (wie immer in PASCAL) absolut keine Bedeutung. Dies geschah nur, damit sie besser auffallen.

Auf oberster Ebene, also da, wo sonst das Hauptprogramm steht, hat das Modul einen leeren Anweisungsteil. Er hätte auch - wie oben erwähnt - ganz weggelassen werden können. Auch haben Sie die Wahl, ob Sie am Ende dieses Pseudo-Hauptprogramms einen Punkt, ein Semikolon (wie oben geschehen) oder auch gar nichts setzen. Es ist nicht möglich, zwischen dieses "Begin End" irgendwelche Befehle zu setzen.

Die Syntax eines Moduls unterscheidet sich davon einmal abgesehen also nicht von der eines Programms:

```
"MODULE" Programmkopf
 Deklarationsteil
 ["BEGIN" [";"] "END" [";"|"."]]
```

Syntax für "Programmkopf":

```
Bezeichner  [ "(" Bezeichner { "," Bezeichner } ")" ]  [ ";" ]
```

Man darf also auch hinter dem Modulnamen (der keine Bedeutung hat) eine Parameterliste wie "(input,output)" setzen - die dann ebenfalls keinerlei Bedeutung hätte.

Eine Prozedur wird importiert, indem man ihren Kopf hinschreibt und anstelle eines Prozedurblocks einfach "Import" schreibt. Als Export wird sie definiert, indem man zwischen ihrem Kopf und Rumpf den Bezeichner "Export" setzt. Ein- und ausgeführte Variablen werden ganz normal deklariert, nur daß hinter die jeweilige Deklarationszeile "Import" bzw. "Export" zu schreiben ist.

Das Hauptprogramm ruft als erstes die Prozedur "p" mit dem Parameter 20 auf. Diese Prozedur wird aus dem Modul importiert und initialisiert die Variable "i", die sie wiederum aus dem Hauptmodul importiert hat. Die Prozedur "q", die das Hauptprogramm als nächstes aufruft, importiert die Funktion "f" lokal, so daß sie nur innerhalb von "q" bekannt ist. Es ist aber nicht möglich, umgekehrt eine lokale Prozedur oder Funktion zu exportieren.

Es sei noch einmal darauf hingewiesen, daß ein Programm stets aus genau (!) einem Hauptprogram, aber beliebig vielen Modulen besteht.

7.3 Objektdateien und Linkersteuerung

Wenn Sie die obigen Listings so von KICK-PASCAL compilieren lassen, ist der Compiler zwar einverstanden, aber der Linker meldet Fehler, weil die importierten Bezeichner nirgendwo definiert werden. Also müssen Sie ihm irgendwie sagen, was er einlinken soll.

Dazu brauchen Sie zunächst einmal Objektdateien, für jeden Bestandteil des Programms einen. Lassen Sie also das "Unterprogramm"-Modul compilieren und ignorieren Sie die Linker-Fehlermeldungen. Jetzt können Sie weder das Programm ausführen noch eine Exe-Datei schreiben lassen. Eine Objektdatei kann man aber unabhängig vom Linker erzeugen lassen. Tippen Sie nach dem Compilieren also "O" (oder wählen Sie das entsprechende Pull-Down-Menü) und speichern Sie die Objektdatei ab. Diese Datei enthält den Programmcode des Moduls und Angaben über die deklarieren Bezeichner.

Laden Sie nun das Hauptprogramm. Jetzt müssen Sie dem Linker sagen, daß die zuvor erzeugte Objektdatei dazu gehört, und zwar mit der Compiler-Anweisung "link":

```
{$link 'pascal:modul.o'}
```

Diese Anweisung können Sie an eine beliebigen Stelle des Quelltextes von "Haupt" setzen. Wenn der Compiler auf diese Directive trifft (siehe dazu auch Kapitel "Compiler-Directives"), lädt er sofort die entsprechende Datei. Es gibt dafür zwei naheliegende Positionen: Entweder nahe der Stelle, wo die zugehörigen "Import"-Deklarationen stehen (ist übersichtlich, weil man dann sofort sieht, was woher kommt) oder als allerletzte Zeile ganz am Programmende. Letzteres ist während der Programmentwicklung praktisch, weil man in der Regel immer wieder neue Compiler-Fehlermeldungen verursacht, während man beim Linken eigentlich nichts mehr falsch machen kann, sobald Im-/Exporte erst einmal stimmen. Deshalb will man so schnell wie möglich wissen, ob das gerade bearbeitete Modul syntaktische Fehler enthält und wird deshalb die Anweisungen zum (bei Diskettenbenutzung erwähnenswerte Zeit erforderndes) Laden von Objektdateien ans Ende setzen, so daß sie erst ausgeführt werden, wenn sonst alles OK ist.

Wie dem auch sei, nun wird der Compiler irgendwann auf die Disk oder Platte zugreifen und die genannte Objektdatei laden. Nach dem Compilieren prüft der Linker wieder, ob alle irgendwo (im Hauptprogramm oder einem eingelinkten Modul) als Import deklarierten Bezeichner irgendwo exportiert werden. Er gibt dann eine vollständige Liste aller undefinierten Bezeichner aus oder, wenn alles klar ist, "Ready". In letzterem Fall haben Sie ein lauffähiges gelinktes Programm vorliegen. Sie können es wie gewohnt starten oder eine Exe-Datei erzeugen lassen.

Die Benutzung von "$link" ist nicht ganz problemlos, wenn Sie Ihre Programmdateien in ein anderes Verzeichnis kopieren und dort neu compilieren wollen. Dann

müssen Sie in jedem Quelltext vor dem Übersetzen die Dateinamen in den Linker-Anweisungen von Hand ändern.

Um das zu umgehen, bietet Ihnen "$link" noch eine interessante Option: Wenn der Dateiname der Objectdatei keinerlei Pfadangabe enthält (d. h. im Namensstring weder ein ":" noch ein "/" vorkommt), wird der Pfad des Dateinamens der im Editor gehaltenen Datei vor den Objectdateinamen gehängt (Sätze gibt's...). Beispiel: In einem Quelltext namens "PASCAL:Sources/Chaos/Name.p" taucht die Anweisung {$link "Hilfsmodul.o" } auf. Dann lädt der Linker die Objectdatei "PASCAL:Sources/Chaos/Hilfsmodul.o".

Dem Linker ist es übrigens, im Gegensatz zum Compiler, egal was ein Hauptprogramm und was ein Modul ist. Der Compiler bringt ihm das bei, indem er das Hauptprogramm immer automatisch den Bezeichner "_main" exportieren und jedes Modul diesen Bezeichner importieren läßt. Dadurch ist gewährleistet, daß ein Programm nur korrekt linkbar ist, wenn es einerseits mindestens ein Hauptprogramm (sonst wäre der Bezeichner "_main" ja undefiniert) und andererseits auch höchstens ein Hauptprogramm enthält (sonst würde ein Doppel-Export von "_main" als Fehler gemeldet).

Dieses "_main" markiert übrigens den Anfang des Hauptprogramms, und am Anfang des Codes jedes Moduls steht ein "Jmp" (das Äquivalent im Maschinensprache zum "Goto" in PASCAL) zu eben diesem Symbol. Das hat dann wiederum die Konsequenz, daß Sie in obigen Beispiel auch Hauptprogramm und Modul vertauschen könnten, also erst ein Object-File des Hauptprogramms erzeugen und es dann in das Modul einlinken lassen.

In der praktischen Programmierung wird es sinnvollerweise so aussehen, daß Sie Objektdateien von allen Modulen und vom Hauptprogramm anlegen und dann den Quelltext mal der einen, mal der anderen Komponente im Editor bearbeiten. Dabei enthält jeder einzelne Quelltext die Compiler-Directives, die alle anderen Teile einladen. Wenn Sie jeweils einen Text verbessert und ausgetestet haben und an einem anderen Programmteil weiterarbeiten wollen, sollten Sie vor dem Laden des nächsten Quelltexts zuerst das Ergebnis Ihrer Arbeit als Objektdatei abspeichern. Das geht vom Tastaturmenü ja sehr bequem, wenn Sie sich an den vorgeschlagenen Dateinamen halten: Es kostet Sie nur die beiden Tastendrücke "O" und "Return" - und natürlich, falls noch nicht geschehen, auch noch die Taste "S" zum Abspeichern der neuesten Version des Quelltextes.

Eine Objektdatei enthält immer nur den Code des zuvor compilierten Programmteils, nicht etwa auch die mit {$link } eingeladenen Codestücke oder das KICK-PASCAL-Laufzeitsystem. Sie brauchen also beim oben beschriebenen Programmierablauf nicht zu befürchten, daß ein Teil im Code doppelt auftaucht.

7.4 Alink-Kompatiblität - Fluch und Segen

Der integrierte Linker von KICK-PASCAL ist im wesentlichen zum AMIGA-Standardlinker "Alink" bzw. der PD-Version "Blink" kompatibel, wenn er auch einige weniger wichtige Features dieses Linker-Standards nicht unterstützt. Das hat die erfreuliche Konsequenz, daß Sie problemlos Assembler-Routinen einlinken können (siehe übernächstes Kapitel), denn fast alle AMIGA-Assembler erzeugen Alink-kompatible Objektdateien. Aber es gibt auch einen Nachteil: Da der Alink im Prinzip für alle Programmiersprachen verwendet wird, hat er keinerlei Vorstellung davon, was eine Prozedur oder eine Parameterliste ist.

Seine Arbeit beschränkt sich darauf, Bezeichner zuzuordnen: Wenn ein Bezeichner irgendwo benutzt und anderswo definiert wird, trägt er an der ersten Stelle die entsprechende Adresse ein - sonst nichts.

Das bedeutet für Sie als PASCAL-Programmierer, daß Sie selbst auf die modulüberschreitende Semantik achten müssen. Sie könnten zum Beispiel versehentlich in einem Modul eine Variable exportieren und in einem anderen Modul eine gleichnamige Prozedur importieren. Ergebnis: Der Compiler weiß nichts über die anderen Module und meldet deshalb keinen Fehler, der Linker, der keine Ahnung von PASCAL hat, stellt hier eine Namensgleichheit fest. Zur Laufzeit wird das Programm dann mit einen Unterprogrammaufruf in einen Datenbereich springen - ein Fall für den Exception-Handler.

Ein anderes Beispiel: ein Modul exportiert eine INTEGER-Variable (mit 2 Bytes Speicherbedarf), ein anderes importiert diese als 4 Byte lange LONGINT-Variable. Wenn das importierende Modul nun lesend auf die Variable zugreift, ist das Ergebnis undefiniert, und wenn es schreibend zugreift (Wertzuweisung), wird über den für die INTEGER- Variable reservierten Bereich hinaus geschrieben - mit unabsehbaren Folgen.

Auch ist es wichtig, daß beim Im- und Export von Prozeduren und Funktionen die Parameterlisten der Import- und Export-Deklaration äquivalent sind - auch hier können Sie sonst "nette" Überaschungen erleben. Aus all' dem folgt für Sie, daß Sie bei Im- und Exporten aller Art höllisch aufpassen müssen.

Als PASCAL-Programmierer sind Sie es gewohnt, daß "Test", "test" und "TEST" lediglich verschiedene Schreibweisen für den selben Bezeichner sind, denn in PASCAL ist die Groß-/Kleinschreibung ja egal. Nun wurde der Alink ursprünglich für C und Assembler geschrieben, und da ist (bei C immer, bei den AMIGA-Assembler meistens) die Schreibweise eines Bezeichners signifikant. Folglich mag der Alink es nicht, wenn man eine Prozedur "Hallo" exportiert, aber "hallo" importiert - für ihn sind dies völlig verschiedene Bezeichner.

Das ist natürlich nicht schön. Deshalb hat der ins KICK-PASCAL-System integrierte Linker eine (Default-mäßig eingeschaltete) Option, die ihn veranlaßt, die Groß-/Kleinschreibung zu ignorieren. Diese Option wird mit dem Pull-Down-Untermenü "Optionen/Linker" geschaltet. Wenn man sie ausschaltet ("GROSS/klein beachten"

angewählt), müssen die Bezeichner in der IMPORT-Zeile genau so wie in der EXPORT-Deklaration geschrieben werden - wie man ihn im Rest der Module schreibt, ist Sache des Compilers und somit dem Linker egal.

Vielleicht fragen Sie sich jetzt, warum es diese gegen alle lieben Gewohnheiten des PASCAL-Programmierers verstoßende Option überhaupt gibt. Well, vielleicht haben Sie einmal das Bedürfnis, Ihr Programm extern zu linken, z. B. im Zusammenhang mit der Einbindung von Assembler-Routinen. Dann will man keine bösen Überraschungen erleben, sondern schon in der KICK-PASCAL-Entwicklungsumgebung erfahren, ob der externe Linker später Fehler melden wird - also sollte sich der KICK-PASCAL-Linker dann genauso verhalten wie sein Kollege Alink.

7.5 Hunks

Dieser Abschnitt ist für den "normalen" PASCAL-Programmierer nicht unbedingt nötig. Ich werde hier die Einträge erklären, die plötzlich in der Speicherbelegungsliste auftauchen, wenn mit Units oder Modulen gearbeitet wird.

AMIGA-Programmdateien - also z. B. die von KICK-PASCAL erzeugten Exe-Dateien - sind in sogenannten "Hunks" organisiert. Es gibt im wesentlichen drei Hunktypen:

- CODE: Ein solcher Hunk enthält lauffähigen Programmcode.
- DATA: Hier können die Daten eines Programms liegen. In Maschinencode gibt es aber keinen prinzipiellen Unterschied zwischen Programm und Daten, so daß dieser Hunktyp eigentlich mit "CODE" identisch ist.
- BSS: Auch hier liegen Daten, aber solche, die beim Programmstart keinen definierten Inhalt haben. Ein BSS-Hunk ist deshalb eigentlich nicht mehr als eine Anweisung an die Laderoutine, Speicher zu reservieren, in dem das Programm dann z. B. Variablen unterbringen kann.

Für Leute, die gern in Hexdumps wühlen: Ein "CODE"-Hunk beginnt in der Datei mit dem Langwort $000003E9, "DATA" mit $000003EA und die Kennung $000003EB steht für "BSS". Das Langwort nach der Hunk-Kennung enthält jeweils die Hunklänge in Langworten, danach kommt der Inhalt des Hunks (außer natürlich bei "BSS", der ja keinen definierten Inhalt hat). Hinter jedem Hunk kann noch eine Relokationstabelle (Kennung: $000003EC) stehen, die Angaben dazu enthält, wie der Hunkinhalt zu modifizieren ist, um ihn an eine Speicheradresse anzupassen - denn im AMIGA wird ja grundsätzlich nicht mit festen Speicheradressen gearbeitet - und wie die Hunks verbunden sind. Bitte verwechseln Sie "Hunk" nicht mit "Hrung", zumal sowieso keiner weiß, was ein Hrung ist und warum er gerade auf Beteigeuze 7 explodiert ist.

Was hat das Ganze mit KICK-PASCAL und Modulen zu tun? Nun, auch Objektdateien sind aus Hunks aufgebaut, nur daß die Hunks zum Teil nicht über Relokationstabellen, sondern über Verweise auf Bezeichner verbunden sind, so daß der Linker im wesentlichen die Aufgabe hat, gleiche Symbole zu verbinden und daraus Relokationstabellen zu erstellen.

In der Speicherbelegungstabelle, die bei allen passenden und unpassenden Gelegenheiten ausgegeben wird, finden Sie eine Liste der vom Compiler erzeugten oder aus Objektdateien geladenen Hunks: Als erstes wäre da der Code-Hunk, dessen Erzeugung ja Sinn und Zweck eines jeden Compilers ist. In Fehlermeldungen wird dieser Hunk übrigens mit "*.code" bezeichnet. Darunter finden Sie in der Zeile "Reloc" Angaben über die Relokationstabelle dieses Hunks. Die Relocs aller anderen Hunks werden nicht aufgelistet, zumal diese Tabellen Ihnen als Programmierer so ziemlich egal sein können. KICK-PASCAL benutzt hier intern auch ein anderes Tabellenformat als in den Objektdateien.

Wenn Sie mit statischen oder exportierten Variablen arbeiten, legt der Compiler dafür auch einen BSS-Hunk an (siehe nächstes Kapitel). Dieser Hunk hat den Namen "*.bss" und taucht in der Liste, zusammen mit den dazugeladenen Hunks, unter "Link:" auf. Die Hunks der geladenen Objektdateien erscheinen jeweils unter ihrem Dateinamen mit angehängtem Hunktypen ".code", ".data" oder ".bss". KICK-PASCAL erzeugt keine DATA-Hunks, man kann sie aber z. B. beim Einlinken von Assembler-Routinen erhalten.

Noch ein Hinweis zum Schluß: Das AMIGA-Dateiformat bietet auch die Möglichkeit, Hunks mittels einer besonderen Kennung gezielt ins Chip- oder Fastmemory laden zu lassen. Wenn eine Objektdatei mit solchen Hunks eingeladen wird, erscheinen wird an die Hunknamen zusätzlich noch ".c" (für "Chip) oder ".f" (für "Fast") angehängt.

7.6 Dynamische und statische Variablen

Normalerweise verwalten KICK-PASCAL-Programme ihre Variablen auf dem Stack. Beim Programmstart wird dort genug Platz für die globalen Variablen eingerichtet, und immer, wenn eine Prozedur oder Funktion aufgerufen wird, reserviert sie sich als erstes die benötigte Menge Speicher. Durch dieses Variablenkonzept ist es möglich, daß jede Schachtel einer rekursiven Prozedur ihre eigenen Variablen hat.

Der Compiler berechnet, wieviel Stack das Programm ohne Rekursion maximal (d. h. zum Zeitpunkt der maximalen Schachtelungstiefe) braucht, und gibt diesen Speicherbedarf als "Linear stack requirement" aus.

Daneben gibt es noch ein weiteres Variablenverwaltungskonzept: Statische Variablen. Solche Variablen liegen in einem BSS-Hunk, sind also Bestandteil der Exe- oder Objektdatei und werden somit schon beim Laden des Programms eingerichtet. Jede statische Variable existiert dadurch immer genau einmal, auch lokale Variable einer rekursiven Prozedur.

Exportierte Variable sind immer statisch, denn der Linker muß ja schon nach dem Compilen wissen, wo sie zur Laufzeit im Speicher liegen, was bei dynamisch auf den Stack gelegten Variablen natürlich nicht möglich wäre. Ferner sind die "globalen" Variablen in Modulen (also die Variablen, die dort außerhalb einer Prozedur oder Funktion deklariert werden), stets statisch. Grund: Um sie auf dem Stack zu

verwalten, müßte jedes Modul zur Compilezeit "wissen", wie viel Platz alle anderen Module für ihre Variablen brauchen und wo noch Platz für die eigenen frei ist. Von diesen inhärent statischen Variablen merken Sie in der Regel nichts, außer, daß die Speicherbelegungsliste einen entsprechend großen BSS-Hunk ausweist oder daß Sie Schwierigkeiten bekommen, falls Sie auf die Idee kommen sollten, eine lokale Variable einer rekursiven Prozedur zu exportieren. Sie können Variablen aber auch bewußt als statisch deklarieren: mit der Directive "STATIC". Sie wird genau wie "Import" oder "Export" hinter die Variablendeklaration gesetzt, z. B. so:

```
Program Statisch;

Var v: integer;

Procedure p(flag: Boolean);
   Var v: integer; STATIC;
   Begin
     If flag Then v:= 26731
             Else writeln(v)
   End;

Begin
   v:= 42;
   p (true);
   p (false);
   writeln (v)
End.
```

Dieses Programm erzeugt die Ausgabe:

```
26731 42
```

Wie funktioniert es? Die Prozedur "p" hat eine lokale Variable "v", die nichts mit der gleichnamigen globalen Variablen zu tun hat. Beim ersten Aufruf mit dem Parameter "true" wird diese lokale Variable mit dem Wert 26731 initialisiert. Dadurch aber, daß "v" statisch ist, existiert es nach dem Ende der Prozedur weiter und hat beim zweiten Aufruf immer noch den selben Wert - andernfalls wäre er undefiniert.

Die globale Variable "v" habe ich im Beispiel eingeführt, um zu zeigen, daß das statische "v" zwar zur gesamten Laufzeit existiert, aber trotzdem nicht global ist.

Übrigens: Wenn Sie das "STATIC" wegließen, würde sich in diesem Fall nichts an der Ausgabe ändern, denn zur dynamischen Erzeugung des lokalen "v" wird immer derselbe Speicherbereich benutzt. Zwischen den beiden Aufrufen von "p" wäre dieser Speicher aber ungeschützt und akut überschreibgefährdet.

7.7 Externes und internes Linken mit "PasLib.o"

Auf Ihrer KICK-PASCAL-Disk finden Sie eine Datei "paslib.o". Sie enthält das gesamte Laufzeitsystem von KICK-PASCAL, also eine Sammlung von Unterprogrammen wie z. B. Ein- und Ausgabeoperationen aller Art, die dem PASCAL-Programmierer zur Verfügung stehen, der Prozessor aber nicht ohne weiteres selbst kann. Beispielsweise ist eine Addition zweier Integer-Zahlen als ein einziger Maschinensprachbefehl zu realisieren, während die Ausgabe einer solchen Zahl eine sehr komplexe Operation und nur als ein längeres Unterprogramm realisierbar ist.

Ein identisches Laufzeitsystem ist integrierter Bestandteil des KICK-PASCAL-Systems, so daß der Linker es ohne Diskettenzugriff mit dem vom Compiler erzeugten Code und den evtl. zugeladenen Modulen zusammenbinden kann. Die Datei "paslib.o" brauchen Sie deshalb nur, wenn Sie einen anderen als den integrierten Linker benutzen, z. B. um mit dem KICK-ASS eine Assembler-Unterroutine zu schreiben und aus dem KICK-ASS-System heraus zu debuggen (mehr dazu im folgenden Abschnitt).

Sie können auch "einfach mal so" extern linken: besorgen Sie sich "BLink", erzeugen Sie zu einem PASCAL-Programm eine Objektdatei, z. B. "hello.o", und tippen Sie im CLI:

```
BLink hello.o, paslib.o to Hello
```

Auf diese Weise erhalten Sie aus der Objekt-Datei eine lauffähige Exe-Datei. Nun benötigt wohl kein einzelnes Programm sämtliche Möglichkeiten, die KICK-PASCAL bietet. So wird man in der Regel entweder "REAL" oder "LONGREAL" benutzen, nicht aber beide Zahlenformate durcheinander - falls man überhaupt Fließkomma-Operationen braucht. Um in Exe-Files nicht zu viel unnützen Code stehen zu haben, ist das ca. 10 KByte große KICK-PASCAL-Laufzeitsystem in mehrere "Bibliotheken" unterteilt: Zum einen wäre da die knapp 4 KByte umfassende Grundlibrary. Somit gibt der Linker immer eine "Maximum Code Size" von wenigstens 3900 Byte aus. Diese Bibliothek enthält im wesentlichen die wichtigsten Ein-/Ausgabeoperationen, die Initialisierungs- und die Endprozedur, welche am Anfang bzw. Ende des Programms aufgerufen werden, die meisten Laufzeit-Fehlermeldungen und so fort. Sie exportiert zum Linker das Symbol "_paslibbase", das ihre Basisadresse enthält.

Oberhalb dieser Adresse, die am Programmanfang ins Prozessorregister A5 geladen wird und dort unverändert stehen bleiben muß, stehen verschiedene wichtige Daten wie z. B. der "Global Vector", über den KICK-PASCAL die Stack-Variablen verwaltet; unterhalb der Basis steht eine Sprungtabelle zu den einzelnen Funktionen. Außerdem gibt es noch sechs weitere, nach "Themen" geordnete Bibliotheken, die ebenfalls jeweils ihre Basisadresse exportieren:

_reallibbase:	Fließkommaoperationen, einfache Genauigkeit
_doublelibbase:	Fließkommaoperationen, doppelte Genauigkeit
_setlibbase:	alle SET-Operationen
_stringlibbase:	größere Stringoperationen wie "+", "Insert" usw.
_misclibbase:	diverses, z. B. "Random", "Hi", "ClrEoL", ...
_amigalibbase:	AMIGA-spezifisches (Windows, Messages, Console...)

Welche Konsequenzen hat das für Sie als PASCAL-Programmierer? Zum einen dürfen Sie die oben genannten Bezeichner (und "_main") nicht aus PASCAL-Modulen exportieren, denn sonst kommt es zu Konflikten.

Wichtiger ist aber, daß der interne Linker nur die Bibliotheken des Laufzeitsystems einbindet, aus denen auch Funktionen benutzt werden, während man bei externem Linken nur die Möglichkeit hat, alles einzubinden.

7.8 Units

Das Unit-Konzept ist eine sehr leistungsfähige Form der modularen Programmierung. Hier werden Compiler und Linker durch spezielle Sprachkonstrukte näher aneinander gebunden, wodurch viele Fehlermöglichkeiten, die es bei den MODULEn gibt, ausgeschlossen werden. Außerdem bieten Units eine bequeme Möglichkeit, den Sprachumfang von KICK-PASCAL durch Anlagen von Unit-Bibliotheken nahezu beliebig zu erweitern.

7.8.1 Ein erstes Beispiel

Analog zu den MODULEn besteht ein KICK-PASCAL-Programm aus einem Hauptprogramm und beliebig vielen Units. Hier ist ein erstes Beispiel für ein Unit:

```
UNIT Beispiel1;

  { Ein Unit beginnt mit dem Schlüsselwort "UNIT", gefolgt vom
    Unitnamen. Während Programm- und Modulnamen bedeutungslos sind,
    werden die Unitnamen für mehrere Zwecke benötigt, so daß man
    sie sinnvoll wählen sollte. }

INTERFACE

  { Ein Unit besteht im wesentlichen aus zwei Teilen: dem
    "öffentlichen" (also dem Hauptprogramm und anderen Units
    zugänglichen) Interface-Teil, und dem "privaten" (für andere
    Programmteile verborgenen) Implementation-Teil. }

 TYPE Natuerlich = 0..MaxInt;
  { Dieser Datentyp wird im Interface-Teil des Units "Beispiel1"
    definiert und steht damit allen Programmen, die dieses Unit
    benutzen, zur Verfügung. }

 VAR Grenze: Natuerlich;
  { Dies ist eine Variable des Units "Beispiel1" }
```

```
PROCEDURE Eingabe (VAR Zahl: integer);
  { Diese Prozedur wird vom Unit "Beispiel1" exportiert. Im
    Interface-Teil steht nur ihr Prozedurkopf, der "Pumpf"
    folgt im Implementationsteil. }

FUNCTION IstPrim (n: Natuerlich): Boolean;
  { Auch diese Prozedur wird von diesem Unit exportiert. }

IMPLEMENTATION

  { Hier beginnt der zweite, nicht-oeffentliche Teil des Units. Er
    enthält die Rümpfe der im Interfaceteil deklarierten Prozeduren
    und Funktionen sowie alle Bezeichner dieses Units, die nicht
    exportiert werden sollen. }

  FUNCTION Teilbar (a, b: Natuerlich): boolean;
   { Diese Funktion gibt an, ob a durch b teilbar ist. }
   BEGIN
      Teilbar:= (a MOD b = 0)
   END;

  FUNCTION IstPrim;
   { Jetzt kommt der Pumpf dieser im Interfaceteil deklarierten
     Funktion. Beachten Sie, daß der Funktionskopf hier keinerlei
     Parameter und keinen Ergebnistypen enthält, denn das würde
     alles ja schon im Interfaceteil definiert.
     Die Funktion soll ermitteln, ob die naturliche Zahl "n" eine
     Primzahl ist. }
   VAR Zaehler: integer;
   BEGIN
     Zaehler:= 2;
     WHILE (Zaehler <= n DIV 2) AND NOT Teilbar(n, Zaehler) DO
     Zaehler:= Zaehler+1;
     IstPrim:= NOT Teilbar(n, Zaehler)
   END;

  PROCEDURE Eingabe;
   { Auch hier wird die Parameterliste nicht noch einmal geschrieben }
   BEGIN
     Write('Bitte eine Zahl eingeben: ');
     Readln (Zahl)
   END;

END. { Dies ist das Ende des Units. }
```

Wenn der Compiler dieses Unit fehlerfrei übersetzt hat (hier können Sie es ignorieren, wenn der Linker Fehler meldet, denn ein Unit kann man sowieso nicht direkt starten), legt er automatisch zwei Dateien an: Die Objectdatei "Beispiel1.o", die wie bei Modulen den von Compiler erzeugten Code des Units enthält, und die Interface-Datei "Beispiel1.u". Darin steht in vorcompilierter Form eine Kopie des Interface-Teils des Units.

Aber wohin werden diese Dateien geschrieben? Im Pull-Down-Menü "Optionen" finden Sie den Menüpunkt "Suchpfade". Wenn Sie diesen mit der Maus anwählen, erscheint ein kleines Fenster mit je zwei Textgadgets für Includefiles und für Unit-Dateien. Hier interessieren uns nur die beiden Zeilen unter "Unit".

Falls Sie nur die erste Zeile benutzen, ist alles einleuchtend: Dann werden die beiden oben genannten Dateien in dieses Verzeichnis geschrieben. Wenn Sie zwei Pfade angeben, wird wieder derselbe raffinierte Trick wie bei den Include-Files angewendet: Die Dateien werden doppelt erzeugt, und zwar in den beiden angegebenen Verzeichnissen. Nun werden die Dateien ja nicht "einfach so" erzeugt, sondern sollen auch irgend wann einmal gelesen werden (dazu später mehr). Zuerst sucht das PASCAL-System im ersten und dann im zweiten angegebenen Verzeichnis nach den Dateien. Wenn sie nur im zweiten Verzeichnis gefunden werden, werden sie ins erste umkopiert (nicht umgekehrt!). Also können Sie als ersten Pfad ein Unterverzeichnis in der schnellen RAM-Disk und als zweiten ein Directory auf Ihrer Festplatte bzw. PASCAL-Arbeitsdiskette wählen. Bequemer ist es natürlich, wenn Sie die Pfade nicht nach jedem Start des KICK-PASCAL-Systems neu eintragen, sondern mit dem Programm "PASCALPrefs" die richtige Einstellung in Ihr Config-File eintragen.

Übrigens werden bei der Unit-Handhabung wie beim Laden von Includefiles Unterverzeichnisse bei Bedarf vom PASCALsystem automatisch erzeugt.

7.8.2 USES

Nun haben wir also ein Unit in Form von zwei bzw. vier Dateien vorliegen. Jetzt können wir es ganz einfach in einem unserer Programme benutzen, z. B. so:

```
PROGRAM Primzahl;

USES Beispiel1;

VAR i: Natuerlich;

BEGIN
   Eingabe (i);
   IF IstPrim (i) THEN
     Writeln (i, ' ist prim.')
   ELSE
     Writeln ('Keine Primzahl.')
 END.
```

Sobald der Compiler auf die "USES"-Anweisung trifft, liest er die Datei "Beispiel1.u" wie ein Include-File ein. Dadurch stehen im folgenden die Bezeichner "Natuerlich", "Grenze" (wird hier nicht benutzt), "Eingabe" und "IstPrim" zur Verfügung. Dabei werden Variablen, Prozeduren und Funktionen automatisch als "Import" behandelt. Außerdem veranlaßt die USES-Anweisung den Compiler, zum Schluß noch die Datei "Beispiel.o" einzuladen.

7.9 Units - Alles noch 'mal etwas genauer

7.9.1 Syntax

Ein Unit beginnt immer mit dem Wort "UNIT", gefolgt vom Unit-Namen, der ein Bezeichner ist. Dahinter beginnt automatisch der Interface-Teil. Sie können dies dadurch deutlich machen, daß Sie das Wort "Interface" schreiben. Dies kann aber auch ersatzlos entfallen.

Danach können USES-Zeilen folgen, denn ein Unit kann selbst wieder andere Units benutzen. Diese Zeilen haben folgende Syntax:

```
Uses-Deklaration = [ "FROM" Bezeichner ] "USES" Bezeichner { "," Bezeichner } ";" .
```

Ein Beispiel:

```
USES Intuition, Graphics; USES Beispiel;
```

Was es mit dem "FROM" auf sich hat, werde ich später verraten.

Nun folgt der Rest des Interface-Parts. Hier kann folgendes stehen:

- Konstanten
- Datentypen
- Variablen
- Librarys
- Prozedur- und Funktionsköpfe

Wie üblich sind Sie an keine feste Reihenfolge gebunden. Labels können hier nicht deklariert werden.

Also hat der Interfaceteil eines Units diese Syntax:

```
Interfaceteil =
  [ "INTERFACE"] Uses-Deklaration { Konstantendefinition |
  Typdefinition | Variablendeklaration | Librarydeklaration |
  Prozedurkopf | Funktionskopf } .
```

Intern behandelt der Compiler die hier deklarierten Funktionen und Prozeduren wie FORWARD-Deklarationen. Sie können aber auch als IMPORT bzw. EXTERNAL deklariert werden, aber nicht als EXPORT, denn sie werden ja sowieso exportiert.

Mit dem Wort "IMPLEMENTATION" beginnt dann der andere Teil des Units. Er ist wie ein normaler Deklarationsteil eines Blocks aufgebaut und endet entweder mit einen "END" oder dem Initialisierungsteil (dazu später mehr). Als letztes darf (!) ein Punkt oder ein Semikolon stehen:

```
Implementationsteil =
  "IMPLEMENTATION" Deklarationsteil ("END" | Anweisungsteil) "." | ";" .
```

Damit hätten wir auch schon die vollständige Syntax für ein Unit:

```
Unit = "UNIT" Bezeichner [ ";" ] Interfaceteil Implementationsteil
```

Im Implementationsteil müssen alle Prozeduren und Funktionen, deren Kopf im Interfaceteil deklariert wurde, definiert werden. Die Reihenfolge der einzelnen Prozeduren ist dabei egal. Hier besteht ihr Kopf aber nur noch aus dem Schlüsselwort PROCEDURE bzw. FUNCTION und dem Namen, die Parameterliste muß (und darf) nicht noch einmal angegeben werden.

7.9.2 Units und Module

Units können von Programmen, Modulen und anderen Units mittels USES aufgerufen werden. Zwei oder mehr Units dürfen sich aber nicht gegenseitig aufrufen. Es besteht also teilweise eine Art Hierarchie unter den Units eines Programms, während MODULEs als gleichrangig behandelt wurden.

Ein Unit-Aufruf aus einem MODULE könnte so aussehen:

```
Module Modul;

Uses Crt;

Procedure Ausgabe; Export;
   Begin
     writeln('Hallo!');
     delay(50)
   End;

Begin End     { Ende des Moduls }
```

Dieses Modul benutzt also das Standard-Unit "Crt", das besondere Prozeduren und Funktionen für die Bildschirm-Ein-/Ausgabe definiert. Ein geeignetes Hauptprogramm zu diesem Modul könnte wie folgt aussehen:

```
Program Haupt;

Uses Crt;

Procedure Ausgabe; Import;

Begin
   Ausgabe
End.

{$link "Modul.o" }
```

Obwohl das Hauptprogramm das Unit "Crt" nicht direkt benutzt, muß in diesem Fall die entsprechende USES-Anweisung gesetzt werden. Dadurch weiß der Compiler, daß er den Code dieses Units laden und dessen Initialisierungsteil (dazu später mehr) aufrufen muß.

7.9.3 Schachtelung von USES-Aufrufen

Nun wollen wir uns noch ansehen, was genau der Compiler beim Aufruf von Units macht. Dazu betrachten wir einige Sourcefragmente:

```
Unit A;
Interface
{ Definitionen von Unit A }
Implementation
End.

Unit B;
Interface
Uses A;
{ Definitionen von Unit B }
Implementation
End.

Unit C;
Interface
Uses B;
{ Definitionen von Unit C }
Implementation
End.

Unit D;
Interface
Uses C,A;
{ Definitionen von Unit D }
Implementation
End.

Program Hauptprogramm;
Uses B,D;
Begin { Hauptprogramm }
End.
```

Als erstes müssen wir hier das Unit A compilieren, da alle anderen Programmteile direkt oder indirekt davon Gebrauch machen. Das Unit B benutzt nur Unit A und kann also direkt danach übersetzt werden.

Erst jetzt kann Unit C compiliert werden. Dabei geschieht folgendes: Durch "Uses B" wird der Compiler angewiesen, den Interface-Teil von Unit B zu übersetzen. Dort steht als erstes "Uses A", so daß auch noch der Interfaceteil von Unit A und erst dann der Rest vom Interface des Units B gelesen wird. Dann wird das Unit C zu Ende übersetzt, wobei dem Compiler die Interface-Deklarationen der Units A und B bekannt sind.

Interessant wird es nun beim Unit D, das erst C und dann A mittels "Uses" benutzt. Zuerst wird Unit C aufgerufen, welches Unit B, das wiederum Unit A benutzt, voraussetzt. Es werden durch die Anweisung "Uses C" also nacheinander die

Interfaceteile von A, B und C (in dieser Reihenfolge) gelesen. Nun geht es weiter im Unit D: es soll zusätzlich noch Unit A benutzt werden. Nun merkt der Compiler aber, daß A schon gelesen wurde und liest es deshalb nicht noch einmal. Somit ist "Uses A" hier überflüssig. (Es hat aber doch eine Wirkung: Das Unit A wird an die erste Stelle der Unit-Liste gesetzt, was wichtig sein kann, wenn mehrere Units gleichlautende Bezeichner definieren. Dazu im nächsten Abschnitt mehr.)

Zuletzt betrachten wir noch das Hauptprogramm. Es benutzt die Units B und D und damit - wie wir bereits wissen - indirekt auch die beiden anderen. Zuerst liest es also das Unit A, um B lesen zu können. Bevor danach D ge-used wird, muß nur noch C gelesen werden, denn A und B sind ja bereits bekannt. Die Reihenfolge der benutzten Units ist also A-B-C-D, genau die selbe, als wenn man im Hauptprogramm nur "Uses D" geschrieben hätte. Somit ist die "Uses B"-Anweisung hier völlig überflüssig.

7.10 Bezeichner-Handhabung in Units

Units sind wunderbar dazu geeignet, den Befehlsumfang von KICK-PASCAL den eigenen Bedürfnissen entsprechend zu erweitern: Man schreibe sich ein Unit mit den neuen Prozeduren, Funktionen, Datentypen etc. und übersetze es einmal - schon kann man es in jedes Programm mit einer einzigen USES-Anweisung einbinden und hat die exportierten Bezeichner dieses Units zur Verfügung. Man kann sogar Units an andere Programmierer weitergeben (sogar verkaufen...), ohne die Quelltexte herausgeben zu müssen: es genügt, wenn der Benutzer die Dateien "MeinUnit.u" und "MeinUnit.o" bekommt.

Stellen Sie sich aber einmal folgendes vor: Von einem Freund haben Sie ein Unit "Matrix" bekommen, das Prozeduren für Matrixoperationen enthält. Auf einer PD-Diskette fanden Sie ein Unit "LongMath", das Prozeduren für das Rechnen mit beliebig großen natürlichen Zahlen definiert. Da Sie ein Mathematik-Freak sind, wollen Sie nun in einem Programm beide Units benutzen. Beim Durchlesen der Dokumentationen zu den beiden Units erschrecken Sie: beide Units definieren eine Prozedur oder Funktion namens "Mult", nämlich für die Multiplikation von Matrizen bzw. langen Zahlen. Was sollen Sie jetzt tun, zumal Sie von keinem der Units den Quelltext haben, die Namen der Prozeduren also nicht ändern können?

Ich kann Sie beruhigen: das Ganze ist kein Problem, denn Units dürfen ohne weiteres identische Bezeichner exportieren. Sie können die beiden oben genannten Units also etwa so in Ihr Programm einbinden:

```
PROGRAM Mein_geniales_Mathematikprogramm;

USES Matrix,     { Geschrieben von Kasimir Müller, Juli 1990   }
     LongMath;   { Gefunden auf der PD-Disk "Quickstart #0815" }
```

Nun bezieht sich der Bezeichner "Mult" auf das als letztes geladene Unit, also "LongMath". Das ist natürlich noch immer nicht schön, denn im Zweifelsfall wollen Sie in Ihrem Programm sowohl Matrizen als auch große Zahlen multiplizieren.

Dafür gibt es in KICK-PASCAL ein besonderes Sprachkonstrukt: Qualifizierte Bezeichner.

Ein Bezeichner wird "qualifiziert", indem man ihn den Namen des Units, in dem er definiert wird, voranstellt - mit einem Punkt getrennt. Also können Sie in "Ihrem genialen Mathematikprogramm" mit "Matrix.Mult" und "LongMath.Mult" gezielt auf die beiden Prozeduren zugreifen.

Globale Bezeichner von Programmen können durch Voranstellen des Programmnamens "qualifiziert" werden, und die vordefinierten Standardbezeichner von KICK-PASCAL bilden ein Pseudo-Unit namens "System". Also kann man durch selbstquälerische Qualifizierungen sogar Standardbezeichner überdefinieren und trotzdem benutzen, wie es im folgenden Beispiel mit der Prozedur "Writeln" geschieht:

```
PROGRAM Qual;

VAR Writeln: integer;

BEGIN
   Qual.Writeln:= 42;
   System.Writeln(Qual.Writeln)
END.
```

Es dürfte problemlos konsensfähig sein, daß qualifizierte Bezeichner nicht gerade das Gelbe vom Ei sind, zumal sie zusätzlichen Schreibaufwand erfordern, was Freaks sowieso vermeiden, wo sie nur können. Deshalb sollten sie eine Notlösung in Fällen wie dem oben beschriebenen sein, während man bei selbstgeschriebenen Units darauf achten sollte, daß die exportierten Bezeichner nicht mit denen anderer Units kollidieren.

Für Leute, die alles ganz genau wissen wollen, soll jetzt noch erläutert werden, wie der Linker mit Units umgeht. Bekanntlich legt der Compiler für jedes Unit zwei Dateien an: Die Interface-Datei mit der Endung ".u" und die Objektdatei, zu erkennen an der Endung ".o".

Erstere wird von Compiler gelesen, sobald er auf eine entsprechende USES-Anweisung stößt, letztere wird anschließend vom Linker in das Hauptprogramm eingebunden. Dabei werden wie bei den Modulen Bezeichner an den Linker übergeben, und zwar alle Namen der Variablen, Prozeduren und Funktionen des Interfaceteils - die dort definierten Konstanten, Datentypen und Librarys sind ja nur für den Compiler von Interesse.

Wenn Sie die Kapitel über Module aufmerksam gelesen haben, wissen Sie vielleicht noch, daß der Linker doppelte Deklarationen gleichnamiger Bezeichner als Fehler meldet. Dagegen können, wie oben beschrieben, verschiedene Units durchaus denselben Bezeichner exportieren. Um dieses Problem zu umgehen, werden die Bezeichner "qualifiziert" an den Linker übergeben. Im obigen Beispiel (dem "genialen" Matheprogramm) heißt das, daß an den Linker die beiden Bezeichner "Matrix.Mult" und "LongMath.Mult" übergeben werden.

Bekanntlich können Sie über ein Pull-Down-Menü einstellen, ob der Linker die Groß-/Kleinschreibung von Bezeichnern beachten soll. Falls Sie nun den weniger toleranten Modus wählen, können Sie eine Überraschung erleben, wenn Sie in der Uses-Zeile die Unitnamen falsch schreiben, z. B. "USES matrix": Der Compiler erfährt dann zwar aus dem Interfacefile, daß die zu importierende Prozedur "Mult" heißt, glaubt aber fälschlich, das Unit heiße "matrix". Also wird versucht, in das Hauptprogramm den Bezeichner "matrix.Mult" zu importieren - exportiert wird aber lediglich der Name "Matrix.Mult". Also meldet der Linker einen undefinierten Bezeichner.

Aus demselben Grunde ist es auch nicht möglich, einem compiliert vorliegenden Unit einen anderen Namen zu geben: Sie können zwar auf der Diskette die Dateien umbenennen, etwa von "Matrix.*" in "Mat.*", und diese Units dann mit "USES Mat" benutzen. Die in der Datei "Matrix.o" exportierten Bezeichner tragen aber weiterhin Namen wie "Matrix.xxx", während Ihr Programm nun "Mat.xxx" importieren will. Ergo meldet der Linker massenhaft Fehler.

7.11 Initialisierung und andere Spezialitäten

7.11.1 Initialisierungs-Prozeduren

In der formalen Syntax von Units wurde bereits angedeutet, daß man Units mit so etwas wie einem Hauptprogramm versehen kann.

Beispiel:

```
UNIT Beispiel2;

INTERFACE

VAR n: integer;

PROCEDURE Ausgabe;

IMPLEMENTATION

PROCEDURE Ausgabe;
   BEGIN
     Writeln ('n = ', n)
   END;

BEGIN    { Initialisierungsteil }
   Write ('n wird initialisiert: ');
   n:= 99;
   Ausgabe
END.
```

Ein Hauptprogramm dazu könnte so aussehen:

```
PROGRAM Hupf;

USES Beispiel;

BEGIN
   n:=n+1;
   Write('Neuer Wert von n: ');
   Ausgabe
END.
```

Das Unit hat also am Ende einen globalen Anweisungsteil, der mit einem "END" endet (sonst enden Units ja mit einen "END" ohne vorangehenden Anweisungsteil). Diese Anweisungsfolge wird der Initialisierungsteil des Units genannt und automatisch vor dem Start des Hauptprogramms, das dieses Unit benutzt, ausgeführt. Also erzeugt unser kleines Programm folgende Ausgabe:

```
n wird initialisiert: n =  0
Neuer Wert von n: n =  100
```

Intern wird das so gehandhabt, daß der Init-Teil als Prozedur mit dem qualifizierten Namen "_init" an den Linker exportiert wird, in unserem Beispiel also "Beispiel2._init". Deshalb können Sie im Interfaceteil eines Units keine Prozedur, Funktion oder Variable mit dem Namen "_init" deklarieren, denn dann würde der Linker ja eine Bezeichnerkollision feststellen und als Fehler melden.

Der Aufruf von Initialisierungsteilen erfolgt im Zweifelsfall genau so, wie man sich das wünscht. Ich möchte nur deshalb genauer darauf eingehen, damit Sie nicht sagen können, das sei nirgendwo dokumentiert:

1. Der Initialisierungsteil jedes direkt oder auch indirekt benutzten Units eines Programms wird genau einmal aufgerufen, auch dann, wenn ein Unit mehrfach benutzt wird (indem es von mehreren benutzten Units ge-used wird).
2. Benutzt ein Unit A ein anderes Unit B, so sorgt KICK-PASCAL dafür, daß in jedem Fall der Initialisierungsteil von Unit B vor dem von Unit A aufgerufen wird. Sie können also beim Programmieren eines Init-Teils davon ausgehen, daß alle Units, die Sie in Ihrem Unit benutzen, bereits initialisiert sind.
3. Von 2. einmal abgesehen, werden Units in der Reihenfolge winitialisiert, in der sie in der USES-Anweisung stehen.

7.11.2 Prüfsummen

Wie Sie bereits wissen, wird der Compiler durch eine USES-Anweisung veranlaßt, die Interface-Datei des entsprechenden Units zu lesen. Dort erfährt er, welche Bezeichner in das gerade übersetzte Unit oder Programm importiert werden, und vor allem auch, welche Bedeutung diese Bezeicher haben. So weiß er nach "USES Beispiel2" (siehe oben), daß "n" eine Integer-Variable und "Ausgabe" eine Prozedur ohne Parameterliste ist. Mit diesem Wissen wird nun der Rest des Programms übersetzt. So weit, so hoopy. Aber jetzt kommt's:

```
UNIT A;

VAR n: LongInt;

IMPLEMENTATION

BEGIN
   n:= 26731
END.
```

Dieses kleine Unit werde nun von einem zweiten benutzt:

```
UNIT B;

USES A;

PROCEDURE Ausgabe;

IMPLEMENTATION

PROCEDURE Ausgabe;
   BEGIN
     Writeln(n)
   END;

END.
```

Dies alles ist noch völlig klar und einsichtig: Das Unit exportiert eine Prozedur, die eine vom Unit "A" importierte LongInt-Variable ausgibt. Aber jetzt (die Spannung steigt ins Unerträgliche...) kommt das Hauptprogramm dazu:

```
PROGRAM C;

USES B;

BEGIN
   Write('Wert von n: ');
   Ausgabe
END.
```

Auch jetzt haben wir noch nicht den springenden Punkt erreicht. Der kommt erst, wenn wir das Unit "A" ändern: Wir deklarieren die Variable "n" im Interface-Teil nicht mehr als LongInt, sondern nur als einfaches Integer, und compilieren dieses Unit mit dieser Änderung.

Und nun (und damit wären wir endlich beim Thema) versuchen wir erneut, unser Hauptprogramm zu übersetzen.

"Na und?"

Das hüpfende Komma (oder so ähnlich) ist, daß das Unit "B" nach der Änderung von "A" noch nicht wieder neu übersetzt wurde. Ergebnis: der vom Compiler erzeugte Maschinencode jenes Units glaubt immer noch, "n" sei eine LongInt-Variable. Resultat: Die Prozedur "Ausgabe" liest ein Langwort (LongInt = 4 Bytes) aus dem Speicher und gibt es aus, obwohl dort nur ein Wort (Integer = 2 Bytes) für die Variable reserviert und initialisiert sind. Ergebnis: Es wird ein undefinierter (also im Zweifelsfall unsinniger) Wert ausgegeben, und Sie werden bei der Suche nach dem vermeintlichen Bug halb wahnsinnig (falls wir einfach 'mal so tun, als ob die Units

und das Hauptprogramm jeweils so ungefähr tausend Zeilen lang und entsprechend unübersichtlich wären).

Aber keine Panik: das Ganze passiert nämlich absolut NICHT! KICK-PASCAL ist clever genug, das Problem zu erkennen, und meldet beim Compilieren des Programms "C" in der USES-Zeile:

```
Checksum Error: Unit "B" using different "A".
```

Alles klar: Das Unit "B" benutzte, als es compiliert wurde, ein Unit "A", das sich von dem jetzt vorliegenden unterscheidet. Nun wissen wir also, daß das Unit "B" neu compiliert werden muß. Am bequemsten benutzt man dazu das Pull-Down-Menü "Starten/Compile Datei" bzw. das Tastaturmenü "eXternal".

Obige Fehlermeldung enthält auch einen Hinweis darauf, woran der Compiler den Fehler benerkt hat: Er legt zu jedem Unit eine Prüfsumme an. Dann werden im Interfacefile jedes Units die Prüfsummen aller benutzten anderen Units vermerkt. Beim Lesen des Interfaceteils von Unit "B" stellte KICK-PASCAL fest, daß das hier benutzte Unit "A" nach dem letzten Übersetzen von Unit "B" verändert wurde.

Die Prüfsumme (32 Bit breit) wird übrigens nur über den Interface-Teil eines Units gebildet. Es hätte also keine Folgen gehabt, wenn Sie nur im Implementationsteil von Unit "A" etwas geändert hätten - denn das ist ja sowieso Privatsache dieses Units. Ferner gehen nur die wesentlichen Teile des Interfaceteils in die Prüfsumme ein, also keine Kommentare, Leerzeilen und andere Layout-Details.

7.12 Projekte - Modulares Programmieren mit Units

7.12.1 Aufräumen im Unit-Verzeichnis

Units sind, wie Ihnen nicht entgangen sein dürfte, ein sehr leistungsfähiges Konzept, das Sie sicher oft benutzen werden. Auf die Dauer wird sich da einiges in Ihrem Unit-Verzeichnis ansammeln: Standard-Units, selbstgeschriebene Units, diverse Units für alle möglichen Zwecke, die Sie auf Ihren KICK-PASCAL-Disketten oder in anderen Quelltextsammlungen gefunden haben usw. - und zwar jeweils zwei Dateien pro Unit. Somit dürfte Ihr Unit-Verzeichnis im Laufe der Zeit ganz schön voll werden. Irgendwann riskieren Sie dann auch, die Übersicht zu verlieren und einem Unit einen bereits vorhandenen Namen zu geben, wodurch das andere Unit überschrieben würde.

Was können Sie also gegen das drohende Chaos unternehmen (einmal davon abgesehen, daß Sie nicht mehr benötigte Units löschen können und sollen)? Seit dem Dahinscheiden von CP/M (die älteren unter Ihnen werden sich vielleicht noch daran erinnern - auch ich habe z. B. auf einem unter CP/M laufenden AppleII-Clone PASCAL gelernt, damals, in der guten alten Zeit... aber lassen wir das!) bietet jedes real existierende Betriebssystem (die 1541 und andere immer noch produzierte Fossilien übergehen wir hier einmal - so etwas KENNT man als AMIGA-User gar nicht) eine Möglichkeit, Verzeichnisse aller Art aufzuräumen: Richtig, Unterver-

zeichnisse sind gemeint ("Sub-Directories" in Basic German). Nichts liegt also näher, als auch im Unit-Verzeichnis damit klar Schiff zu machen...

Das Problem: woher soll KICK-PASCAL wissen, wohin es Units schreiben bzw. von wo lesen soll? (Naive Menschen glauben immer noch, mit einem Computer Probleme lösen zu können. In Wirklichkeit zieht die Anschaffung eines Computers generell unzählige Probleme mit sich, von denen man vorher nicht einmal zu träumen gewagt hätte...) Aber Sie haben Glück: in diesem Abschnitt erfahren Sie nicht nur das Problem, sondern auch die Lösung (Wow!), und zwar genau jetzt (Oh yeah!):

Zunächst müssen wir unser Unit ja in ein Unterverzeichnis schreiben.Dazu dient die Compiler-Directive "$project", die man an beliebiger Stelle im Unit-Quelltext unterbringen kann, z. B. so:

```
{$project MatheTools }
```

Der Projektname muß formal ein Bezeichner sein und ist die Bezeichnung des Unterverzichnisses, in dem das Unit abgelegt werden soll. Dabei wird das entsprechende Verzeichnis automatisch angelegt.

Man kann aber natürlich auch "manuell" in Ihrem Unit-Directory solche Subdirectorys anlegen und Units dort nachträglich ablegen. In Programmen und Modulen darf man "$project" zwar auch verwenden, aber es hat dort keinerlei Wirkung. Als Abkürzung kann man auch "$pro" oder alles, was mit "$proj" anfängt, schreiben.

Um ein Unit aus einem Unterverzeichnis des Unit-Verzeichnisses zu benutzen, setze man einfach ein "FROM <Name>" vor die Uses-Zeile, z. B. so:

```
USES Intuition, Graphics;
FROM MatheTools USES LongMath, Matrix;
FROM Windows USES PASCALCrt;
USES Printer;
```

Mit diesen Zeilen werden die Units "Intuition", "Graphics" und "Printer" aus dem Unit-Hauptdirectory sowie "LongMath", "Matrix" und "PASCALCrt" aus den Unterverzeichnissen "MatheTools" bzw. "Windows" gelesen. Das FROM bezieht sich also immer nur auf das nächste USES.

Für Modula-II-Kenner: Die KICK-PASCAL-Zeile

```
FROM xxx USES yyy;
```

hat nichts mit der Modula-Anweisung

```
FROM xxx IMPORT yyy;
```

zu tun! Ersteres importiert ein ganzes Unit aus einem Subdirectory, letzteres einen einzigen Bezeichner aus einem Modul.

7.12.2 Organisation von Projekten

Units dienen zwei verschiedenen Zwecken: Wie Sie sich Bibliotheken mit öfter benutzten Prozeduren und Funktionen anlegen können, wissen Sie bereits. Aber Sie können mit Units auch modular programmieren, und zwar komfortabler als mit MODULEn - wenn auch mit ein paar kleinen Einschränkungen. Formal besteht natürlich keinerlei Unterschied zwischen "Bibliotheks"- und "Modul"-Units, erstere werden eben von vielen verschiedenen Programmen benutzt, während letztere speziell für ein Programmprojekt geschrieben werden. Trotzdem will ich hier einige Tips für die Realisierung umfangreicher Programmprojekte unter Benutzung von Units geben.

Als erstes sollte man sich einen Namen für das Projekt ausdenken, denn Units, die nur in diesem einem Programm benutzt werden, haben im Unit-Hauptverzeichnis nichts zu suchen. Auch für die Quelltexte ist entweder eine neue Diskette zu formatieren oder ein eigenes Unterverzeichnis anzulegen.

Der nächste Job ist schon etwas anspruchsvoller: Man muß ein gewisses Konzept entwickeln, nach dem man sein Projekt in Units unterteilen will. In der Regel fallen einem bei großen Programmen sofort gewisse Funktionsgruppen auf, die weitgehend unabhängig voneinander arbeiten.

Ein Dateiprogramm könnte man zunächst in folgende Units aufteilen:

- Festlegen der Eingabemaske
- Eingabe eines Datensatzes
- Lesen und Schreiben einer Datei
- Prozeduren zum Drucken von Listen usw.

Jedes dieser Units könnte man noch weiter verfeinern. Das sollte man aber nicht unbedingt übertreiben, z. B. nicht für jede Prozedur ein eigenes Unit. Ein einzelnes Unit sollte zweckmäßigerweise nicht mehr als etwa 1000 Zeilen Quelltext umfassen.

Beim Entwurf dieses Konzeptes ist zu beachten, daß Units sich nicht gegenseitig benutzen können, hier also eine strenge Hierarchie besteht. Deshalb wird man des öfteren mehreren der Hauptunits ein gemeinsames Unterunit zuordnen müssen. Oft ist es sogar zweckmäßig, ein Unit namens "Global" (oder so) zu schreiben, das globale Definitionen (Variablen, Datentypen...) des Programms enthält und von allen anderen Units benutzt wird.

Dann brauchen Sie noch ein Hauptprogramm. Es sollte aus gewissen Gründen (nicht Gewissensgründen) so kurz wie eben möglich sein, im Extremfall beispielsweise bloß ein Unit "Haupt" über USES einbinden und daraus eine Prozedur namens "Hauptprogramm" aufrufen, der ganze Rest könnte dann in jenem Unit passieren.

Ein Programm (in diesem Kapitel heißt das immer: "ein riesiges Programmprojekt") schreibt man natürlich nicht einfach so 'runter. Vielmehr muß es immer wieder getestet, erweitert, getestet, erweitert, getest... werden. Wenn Sie aber versuchen,

ein Unit zu starten, wird es übersetzt, der Code ins Unit-Verzeichnis geschrieben und die monotone Meldung "Can't run a unit!" ausgegeben - auf gut deutsch: Ein Probelauf eines Units ist nicht möglich. Deshalb braucht man das Hauptprogramm, und deshalb sollte es auch möglichst kurz sein: Immer, wenn man ein Unit geändert und compiliert hat, muß man das Hauptprogramm durch den Compiler jagen, so daß man das gesamte Programm mit dem geänderten Unit ausprobieren kann.

Es wäre ein unzumutbarer Arbeitsaufwand, wenn man dazu jedesmal erst den Quelltext des gerade bearbeiteten Units abspeichern und das Hauptprogramm in den Editor laden müßte. Einfacher geht es, wenn man das Hauptprogramm extern übersetzen läßt, also mit dem Menü "Starten/Compile Datei" oder dem entsprechenden Tastaturmenü. Auf diesem Wege kann man übrigens auch Units übersetzen, die wegen eines Prüfsummenfehlers neu compiliert werden müssen.

Noch komfortabler wird es aber, wenn man die Quelltextdatei des Hauptprogramms als "Main File" bzw. "Hauptdatei" definiert. So eine Datei wird immer, nachdem aus dem Editor ein Unit compiliert wurde, extern übersetzt und gelinkt. Also wird nach jeder Änderung eines Units das Gesamtprogramm neu erstellt und kann getestet werden.

7.13 Assembler und PASCAL

Es gibt Tage im Leben eines Programmierers, da kann man es einfach nicht vermeiden, ein Unterprogramm in Assembler zu schreiben; sei es nun aus Geschwindigkeitsgründen oder bei maschinennahen Problemen (versuchen Sie 'mal, einen Exceptionhandler oder Interruptserver in PASCAL zu schreiben...). Der KICK-PASCAL-Linker bindet nicht nur PASCAL-Module zusammen, sondern kann auch Assemblerroutinen einlinken.

Prinzipiell können Sie jeden Assembler benutzen, der Alink-kompatible Objektdateien erzeugt (also fast jeden). Ich empfehle natürlich den KICK-ASS - zum einen, weil ich am Umsatz beteiligt bin, und zum anderen, weil er wirklich sehr bequem ist, z. B. wurde das komplette KICK-PASCAL-System mit dem KICK-ASS programmiert.

Assemblerroutinen werden genau wie PASCAL-Module eingebunden. Sie können auch umgekehrt vom KICK-ASS aus das PASCAL-Hauptprogramm einlinken, indem Sie das entsprechende Objectfile und die Laufzeitbibliothek "paslib.o" laden. Wenn Sie dann noch mit

```
globl _main
```

den Anfang des PASCAL-Programms in das Assemblerprogramm importieren, können Sie mit

```
j _main
```

aus dem KICK-ASS-Monitor das gelinkte PASCALprogramm starten.

Achtung: Als Standard-Ein/-Ausgabe benutzt das Programm das aktuelle CLI-Fenster (der KICK-ASS darf also nicht von der Workbench gestartet worden sein!) und nicht etwa das KICK-ASS-Window.

Und noch ein Hinweis: Es existieren mehrere Versionen des KICK-ASS. Bei den Versionen vor 1.25 ist der Linker nicht so kompatibel, wie er sollte, so daß es bei der Kombination von PASCAL und Assembler unter bestimmten Umständen Probleme gibt.

Ach ja, apropos "globl": Ein Unterprogramm wird aus dem KICK-ASS exportiert, indem man das Label am Anfang der Routine als "globl" deklariert. Auch Variablen kann man auf diese Weise zwischen der Assembler-Routine und dem PASCAL-Programm austauschen.

Ein Programmfragment als Beispiel:

```
Program PASCAL meets Assembler;
Var a, b: integer; EXPORT;
    x: LongInt; IMPORT;
Procedure Assi;    IMPORT;
Begin { Hauptprogramm }
   a:= 47;
   b:= 17;
   Assi;
   writeln(a:10,b:10,x:10)
End.
```

Ein geeignetes KICK-ASS-Programm dazu:

```
        globl a, b, x, Assi, main

        jmp   main              ; Einsprung ins Hauptprogramm

Assi:   move.w  a,d0
        move.w  b,a
        move.w  d0,b            ; Werte von "a" und "b" vertauschen
        muls    a,d0            ; Produkt a*b berechnen
        move.l  d0,x            ; ...und in x ablegen
        rts                     ; zurück ins PASCAL-Programm

x:      blk.l   1               ; Langwort-Variable, wird exportiert
```

Dieses Programm macht nicht viel Tiefsinniges: Zwei INTEGER-Variable werden initialisiert, eine Assembler-Routine, die die Variablenwerte vertauscht und das Produkt in einer LONG-Variablen ablegt, wird aufgerufen, und am Ende werden alle drei Variablenwerte ausgegeben.

Wie kann man nun eine Parameterübergabe realisieren? Es gibt natürlich - wie in obigem Beispiel - die Möglichkeit, die Parameter in gemeinsamen Variablen abzulegen. Wenn Sie einer importierten Routine wie einer PASCAL-Prozedur Parameter übergeben, liegen diese direkt oberhalb der Rückkehradresse auf dem Stack (also ab "4(a7)"), wobei die einzelnen Parameter in ihrer normalen Reihenfolge auf den Stack geschoben wurden, so daß also der erste Parameter an einer höheren Speicheradresse steht als der zweite.

Der Rückgabewert von Functions ist stets im Register d0 zurückzugeben. Ist er vom 64 Bit breiten Typ "LongReal", so ist er in d0 und d1 abzulegen. Der Wert des Adressregisters a5 darf in der Assembler-Routine nicht verändert werden, davon abgesehen stehen Ihnen alle Register zur Verfügung.

Einfacher geht die Parameterübergabe aber, wenn Sie im PASCAL-Programm "External" statt "Import" schreiben. Die Wirkung ist eigentlich dieselbe; wenn aber ein Parameter einen Registernamen wie "d0" oder "a3" trägt, wird der Wert vor dem Unterprogrammaufruf in das entsprechende Register geladen. Bei VAR-Parametern erhält das Register die Adresse der referierten Variablen, bei nichtskalaren Value- Parametern (z. B. Arrays) ist das Ergebnis undefiniert.

Die Parameterübergabe in Registern hat auf die Laufzeit übrigens keine Auswirkung, denn die Parameter werden zuerst wie gewohnt auf den Stack gelegt und erst dann in die Register geladen. Wenn ein formaler Parameter keinen 68000-Registernamen trägt, wird er auch nicht in ein Register geladen, liegt aber wie gewohnt auf dem Stack.

Die Assembler-Unterroutine kann wie ein Modul mit der Anweisung "{$link 'xxx'}" in das PASCAL-Programm eingebunden werden. Dann haben Sie aber möglicherweise Probleme mit dem Pfad, der sich beim Aufräumen auf der Diskette ändern kann. Deshalb gibt es alternativ zu "link" noch die "ulink"-Anweisung.

Beispiel:

```
{$ulink "Assemblerteil.o" }
```

Jetzt wird die angegebene Datei aus dem Unit-Verzeichnis gelesen. Auch hier können Sie Unterverzeichnisse angeben (analog zum "FROM" bei Units):

```
{$ulink "Assemblerzeugs/Test.o" }
```

"ulink" hat noch einen weiteren Vorteil gegenüber "link": Bei Angabe zweier Unit-Pfade kann man mit "ulink" wie bei Units ein Puffern der Objektdateien in der RAM-Disk erreichen. Dann müssen Sie aber aufpassen, wenn Sie gleichzeitig in PASCAL und Assembler programmieren und eine Assembler-Objektdatei ändern: vergessen Sie nicht, die neue Version der Objektdatei in BEIDEN Unit-Verzeichnissen abzulegen, damit KICK-PASCAL sie auch sicher findet.

Stehen "$link" und "$ulink" im Interfaceteil eines Units, so werden sie auch in die Interfacedatei übernommen, so daß die Dateien dann beim "USES"-Aufruf des Units automatisch eingeladen werden.

REFERENZ

KAPITEL VIII

STANDARD-UNITS

VIII. Standard-Units

Auf Ihren KICK-PASCAL-Disketten finden Sie einige fertige Units, die Sie in Ihren Programmen verwenden können. Diese Unit-Sammlung wird ständig erweitert (insbesondere sind Sie, jawohl, genau Sie (!), aufgerufen, von Ihnen geschriebene nützliche Units der Allgemeinheit zur Verfügung zu stellen). Deshalb können wir schlecht alle mitgelieferten Units in diesem Handbuch dokumentieren. Also folgt jetzt eine Beschreibung der wichtigsten Standard-Units, zu den übrigen werden Sie Dokumentationen als ASCII-Datei auf der Source-Diskette finden.

8.1 Crt und PASCALCrt

Das Unit Crt öffnet in seinem Initialisierungsteil auf der Workbench ein Window geeigneter Größe sowie ein Console-Device dazu und verbiegt mit "SetStdIO" die Standard-Ein-/Ausgabe auf dieses Fenster. Also erspart dieses Unit Ihnen viel Routinearbeit, wenn Sie für die Ein- und Ausgaben eines Programms ein eigenes Fenster öffnen wollen.

Außerdem stellt Crt einige Prozeduren und Funktionen zur Verfügung, die Sie vielleicht schon vermißt haben: Mit "ReadKey" und "KeyPressed" ist es endlich möglich, einen Tastendruck abzufragen (beim AMIGA ist dies nicht ohne weiteres über die Standard-EA, die ja ein "CON:"-Window ist oder wenigstens so tut als ob, möglich). Durch die Funktion "WindowClosed" können Sie das Betätigen des Fensterschließsymbols abfragen. Auch andere Ereignisse können Sie bequem abfragen, und da Sie Rastport und Userport des Fensters kennen, können Sie im Ausgabefenster sogar (zusammen mit den Units "Intuition" und "Graphics") Grafik ausgeben oder Gadgets und Menüs verwenden.

Auch wenn Sie alle diese Features nicht brauchen, können Sie das Unit "Crt" benutzen, um Ihre Programme von der Workbench startbar zu machen: "Uses Crt" bewirkt, daß beim Programmstart automatisch ein Window geöffnet und die Standard-EA darauf umgeleitet wird - sonst laufen die Programme genau wie zuvor. Während "Crt" in Assembler programmiert wurde, um den Code möglichst kompakt zu halten, ist "PASCALCrt" (im Unterverzeichnis "Windows") eine sonst vollkommen identische Implementation des Units "Crt" in PASCAL. Dieses Unit liegt Ihnen auch als Quelltext vor, so daß Sie es bei Bedarf modifizieren können.
Es ist aber nicht sinnvoll, in einem Programm sowohl "Crt" als auch "PASCALCrt" zu benutzen, denn dann werden ja gleich zwei identische Ausgabefenster geöffnet.

Folgende Prozeduren und Funktionen werden jeweils in den beiden Units definiert:

Function KeyPressed: Boolean

Die Funktion KeyPressed wird "true", wenn eine Taste gedrückt wird bzw. sich bereits im Tastaturpuffer befindet oder am UserPort des Ausgabefensters eine Nachricht eintrifft bzw. schon anliegt.

Beispiel:

```
Repeat
  Write('Hallo! ')
  Until KeyPressed;
```

Das Programmfragment gibt so lange ununterbrochen "Hallo" aus, bis eine Taste gedrückt oder das Fensterrschließsymbol angeklickt wird.

Procedure WaitForKey

Die Prozedur wartet, bis eine Taste gedrückt wird oder ein Ereignis eintritt. Falls sich bereits eine Taste im Puffer befindet oder eine Nachricht am Window-UserPort anliegt, wird natürlich nicht gewartet.

WaitForKey verbraucht beim Warten keine Rechenzeit, ist also solch' fragwürdigen Konstrukten ("busy wait") wie

```
Repeat Until KeyPressed
```

in jedem Fall vorzuziehen.

Function ReadKey: Char

Diese Funktion liest ein Zeichen von der Tastatur. Dabei wird zuvor durch Aufruf von "WaitForKey" auf eine Taste oder ein Ereignis (Message) gewartet. Falls keine Taste gedrückt, statt dessen aber eine Message empfangen wurde, gibt ReadKey den Wert chr(0) zurück.

Beispiel:

```
Uses Crt;
Var c: Char;
Begin
  Repeat
    c:= ReadKey;
    WriteLn('Ascii-Code', Ord(c):4)
  Until c = chr(13)
End.
```

Das Programm holt Zeichen von der Tastatur und gibt deren ASCII-Code aus, bis der Code der Return-Taste gelesen wird. ReadKey bewirkt kein Echo, d. h. ein gelesenes Zeichen wird nicht automatisch auf den Bildschirm ausgegeben.

Function MessageReceived (Maske): Boolean

Maske: LongInt

Wenn "KeyPressed" wahr wird, kann das zwei Ursachen haben: Entweder wurde eine Taste gedrückt oder am UserPort des Ausgabefensters wurde eine Nachricht empfangen. Diese Messages enthalten Informationen über verschiedene Ereignisse,

z. B. Betätigung des Fensterschließgadgets, Anwahl eines Pull-Down-Menüs usw. Falls eine derartige Nachricht anliegt, liefert "ReadKey" den Wert Chr(0), und man kann mit "MessageReceived" bestimmen, um welche Art von Nachricht es sich handelte.

Im Interfaceteil des Units "Crt" werden die folgenden Konstanten definiert:

```
NEWSIZE          = $00000002; { Änderung der Fenstergröße }
REFRESHWINDOW    = $00000004; { Fensterinhalt neu aufzubauen }
MOUSEBUTTONS     = $00000008; { Drücken/Loslassen einer Maustaste }
MOUSEMOVE        = $00000010; { Mausbewegung }
GADGETDOWN       = $00000020; { Gadget angeklickt }
GADGETUP         = $00000040; { Gadget losgelassen }
REQSET           = $00000080;
MENUPICK         = $00000100; { Menü angewählt }
_CLOSEWINDOW     = $00000200; { Close-Gadget }
RAWKEY           = $00000400; { Tastendruck, Key-Code }
REQVERIFY        = $00000800;
REQCLEAR         = $00001000;
MENUVERIFY       = $00002000;
NEWPREFS         = $00004000; { Ändern der Preferences }
DISKINSERTED     = $00008000; { Einlegen einer Diekette }
DISKREMOVED      = $00010000; { Herausnehmen einer Diskette }
WBENCHMESSAGE    = $00020000;
ACTIVEWINDOW     = $00040000; { Fenster aktiviert }
NACTIVEWINDOW    = $00080000; { Fenster deaktiviert }
DELTAMOVE        = $00100000; { Verschiebung bei Gadgets }
VANILLAKEY       = $00200000; { Tastendruck, ASCII-Code }
INTUITICKS       = $00400000;
```

Diese Zahlen heißen "IDCMP-Flags" ("IDCMP" = "Intuition Direct Communication Message Port").

Man kann mit

```
If MessageReceived(NEWSIZE) Then ...
```

abfragen, ob die Fenstergröße verändert wurde. Da aber jedes Signal einem Bit entspricht, kann man auch abfragen, ob eins von mehreren Signalen empfangen wurde, indem man die Codes mit OR verknüpft:

```
Repeat
   ...
Until MessageReceived(DISKREMOVED Or _CLOSEWINDOW);
```

Diese Schleife wird ausgeführt, bis eine Diskette aus dem Laufwerk entnommen oder das Fenster geschlossen wurde.

Vorher muß man aber dem Betriebssystem mitteilen, von welchen Ereignissen man überhaupt erfahren möchte, und zwar mit den Prozeduren "SetIDCMP" und "AddIDCMP". Als Default ist beim Öffnen des Fensters nur das "_CLOSEWINDOW"-Flag gesetzt. "MessageReceived" holt die Nachricht aber nicht ab - Sie müssen mit der "ReadKey"-Funktion das Chr(0)-Zeichen lesen, damit das Unit "Crt" weiß, daß Sie die Message ausgewertet haben.

Eine Auswertung der Fenster- und/oder Tastaturereignisse könnte also so aussehen:

```
Uses Crt;
Var c: Char;
Begin
  AddIDCMP (MOUSEBUTTONS); { zusätzlich Mausklicks abfragen }
  Repeat
    c:= ReadKey;
    If c = chr(0) Then
      Begin
        { Nachricht auswerten: }
        If MessageReceived(MOUSEBUTTONS) Then
          Writeln(' Klick!');
        If MessageReceived(_CLOSEWINDOW) Then
          Writeln(' Bye bye!');
        { usw. }
      End
    Else
      Begin
        { Taste auswerten, z. B. 3D: }
        Write(c);
      End;
  { Programmende mit Close-Gadget oder Esc-Taste: }
  Until (MessageReceived(_CLOSEWINDOW)) Or (c=#x);

End.
```

Natürlich sind beide Teile optional, d. h. wenn Sie nur Tasten oder nur Messages auswerten wollen, können Sie die jeweils anderen Ereignisse ignorieren. Wichtig ist nur, daß Sie auch zur Message-Behandlung die Funktion "ReadKey" benutzen müssen, denn sonst kann das Unt "Crt" ja nicht wissen, wann Sie eine Nachricht ausgewertet haben und die nächste haben wollen.

Procedure SetIDCMP (Maske)

Maske: LongInt

Diese Prozedur definiert die IDCMP-Flags des Ausgabewindows (siehe unter "MessageReceived"). Als Default ist am Anfang nur "_CLOSEWINDOW" gesetzt.

Procedure AddIDCMP (Maske)

Maske: LongInt

Im Gegensatz zu "SetIDCMP", das die IDCMP-Flags definiert, fügt die Prozedur "AddIDCMP" nur neue Flags hinzu. Der Parameter wird also mit den bisherigen

Flags OR-verknüpft. Somit ist "AddIDCMP(0)" eine leere (nichts bewirkende) Anweisung, während "SetIDCMP(0)" alle Flags löschen würde.

Procedure SubIDCMP (Maske)

Maske: LongInt

Dies ist das Gegenstück zu "AddIDCMP": Die als Parameter übergebenen Flags werden aus der IDCMP-Maske des Fensters gelöscht.

Function WindowClosed: Boolean

Diese Funktion ist eine Abkürzung für "MessageReceived(_CLOSEWINDOW)". Man kann damit bequem abfragen, ob das Close-Gadget des Fensters angeklickt wurde. Es gilt aber dasselbe wie bei "MessageReceived": Die Ereignisse müssen mit der "ReadKey"-Funktion abgeholt werden, damit das "Crt"-Unit weiß, daß sie bearbeitet wurden.

Beispiel:

```
Uses Crt;
 Var c: Char;
 Begin
   Repeat
     c:= ReadKey
   Until WindowClosed
 End.
```

Ohne den "ReadKey"-Aufruf wäre dies eine Endlosschleife, denn die entsprechende Message würde nie am Userport des Windows abgeholt; außerdem wäre es ein "Busy Wait", der Rechenzeit verschwendet.

Function CrtMessage: Ptr

Falls ein Aufruf von "MessageReceived" ergibt, daß eine Nachricht empfangen wurde, liefert diese Funktion einen Zeiger auf dieselbe (sonst "NIL"). Man kann sie dann näher auswerten.

Beispiel:

```
Uses Crt, Intuition;
 Var c: Char;
     m: ^IntuiMessage; { im Unit "Intuition" definiert }
Begin
  AddIDCMP(MOUSEBUTTONS);
  Repeat
    c:= ReadKey;
    If MessageReceived(MOUSEBUTTONS) Then
      Begin
        m:= CrtMessage;
        Writeln('Position:', m^.MouseX:5, m^.MouseY:5);
      End
  Until WindowClosed
End.
```

Wenn Sie mit der linken Maustaste in das Ausgabefenster klicken, gibt dieses Programm die genauen Koordinaten der Position an.

In der "Einführung in die AMIGA-Systemprogrammierung", dem zweiten Teil dieses Handbuchs, steht noch mehr Wissenswertes über diese IDCMP-Messages.

Function CrtWindow: Ptr

Diese Funktion liefert die Windowhandle des Crt-Windows. Sie brauchen Sie für viele Zwecke, z. B. wenn Sie das Fenster mit einem Gadget oder Menü versehen wollen.

Function CrtRastPort: Ptr

Mit dieser Funktion erhalten Sie einen Zeiger auf den RastPort des Crt-Fensters. Diesen Pointer benötigt man vor allem für Grafikbefehle.

Beispiel:

```
Uses Crt, Graphics;
Var Eingabe: Char;
    i: integer;
    x1, x2, y1, y2: Real;
Begin
  Write(#e'0 p');
              { Cursor unsichtbar machen }
  SetAPen(CrtRastPort, 1);
  { Zeichenfarbe: weiß }
  For i:=1 to 200 Do
    Begin
      x1:= 320+300*sin(i/25);
      x2:= 320-300*sin(i/20);
      y1:= 120+100*cos(i/32);
      y2:= 120+100*cos(i/40);
      Move(CrtRastPort, Round(x1), Round(y1));
      Draw(CrtRastPort, Round(x2), Round(y2))
    End;
  Repeat Eingabe:= ReadKey Until WindowClosed
End.
```

Dieses Programm gibt mit Hilfe dreier Funktionen der Graphics-Library (die in Unit "Graphics" definiert wird) ein recht hübsches Muster aus.

Im entsprechenden Abschnitt der AMIGA-Einführung in diesem Buch finden Sie Näheres über die Grafikfunktionen.

Function CrtUserPort: Ptr

Gibt einen Zeiger auf den UserPort des Crt-Fensters.

Function NameOfProgram: Str

Diese Funktion gibt den Namen des Crt-Fensters aus. Im allgemeinen wird hier der Name genommen, unter dem das Programm als Exe-File von der Workbench oder vom CLI geladen wurde. Falls es aber aus der KICK-PASCAL-Entwicklungsumgebung gestartet wurde, gibt es zwei Sonderfälle: wurde KICK-PASCAL aus dem CLI gestartet, erhält das Console-Fenster die entsprechende CLI-Anweisung, beim Workbenchstart wird "KICK-PASCAL" als Default benutzt.

Function CrtConsole: Ptr

Von dieser Funktion erhalten Sie die Handle des Console-Devices, das vom Unit "Crt" für das Ein-/Ausgabefenster geöffnet wird. Wenn Sie in Ihrem Programm ein zweites Window öffnen und mit "SetStdIO" die Standard-EA vorübergehend darauf umleiten, können Sie sie nachher wieder mit "SetStdIO(CrtConsole)" auf das normale Crt-Fenster zurücksetzen.

Function TextLines: integer

Diese Funktion gibt an, wieviele Textzeilen im Ausgabefenster Platz haben.

Function TextColms: integer

Gegenstück zu "TextLines": Gibt die Anazhl der Textspalten des Windows zurück.

Beispiel:

```
USES Crt;
BEGIN
  Writeln('Das Fenster ist ',TextColms,'*',TextLines,' Zeichen
          groß.');
  WaitForKey
End.
```

Function WhereX: integer

Mit dieser Funktion können Sie die aktuelle Cursorposition erfragen. Sie gibt die Spalte zurück, in der der Cursor steht.

Function WhereY: integer

Liefert analog zu "WhereX" die Spalte, in der der Cursor gerade steht.

Procedure WindowTitles (windowname, screenname: Str)

Das Unit "Crt" gibt dem Window den Namen, unter dem das Programm gestartet wurde. Mit der Prozedur "WindowTitles" kann dieser Name sowie der Name der

Workbench geändert werden. Der Titel der Workbench ändert sich aber nur solange, wie das Crt-Fenster aktiv ist.

Wenn Sie einen Parameter auf "Nil" setzen, entspricht dies keinem Namen. Der Wert "Str(-1)" bewirkt, daß der bisherige Name erhalten bleibt.

Beispiele:

```
WindowTitles ('Test', Str(-1))
```

Window heißt "Test", Name des Workbenchscreens bleibt erhalten

```
WindowTitles (Dateiname, 'Editor');
```

Der Fenstername entspricht dem Inhalt der Variablen "Dateiname" (z. B. eine Stringvariable), während der Workbenchscreen bei aktivem Crt-Fenster den Namen "Editor" erhält. Achtung: Wenn Sie den Inhalt der Stringvariablen ändern, wird der Fenstername NICHT automatisch aktualisiert (woher soll das Betriebssystem das auch wissen), sondern erst beim nächsten Refresh der Titelzeile. Sie sollten also bei jeder Zuweisung an die Variable erneut die "WindowTitles"-Prozedur aufrufen.

```
WindowTitles (Str(-1), Nil);
```

Der Fenstername bleibt erhalten, die Workbench wird namenlos.

8.2 Printer

Dieses Unit ist in PASCAL geschrieben und ausgesprochen kurz: Im Interfaceteil wird lediglich eine Text-Dateivariable namens "Lst" definiert und im Init-Teil unter dem Namen "PRT:" zum Schreiben geöffnet. Falls dies nicht gelingt, bricht das Programm mit einer entsprechenden Fehlermeldung ab.

Durch Einbinden dieses Units können Sie also einfach mit

```
Write (Lst, ...)
```

Daten auf den Drucker ausgeben.

8.3 Die AMIGA-System-Units

Zum professionellen Programmieren sind auf dem AMIGA die System- Includedateien unverzichtbar. Sie enthalten zahllose Deklarationen von Konstanten, Datentypen und Library-Funktionen und ermöglichen so den vollen Zugriff auf das AMIGA-Betriebssystem.

Das Problem bei diesen Includefiles ist, daß es ganz schön viele werden können und das Übersetzen somit spürbar Zeit kostet. Um dies etwas zu beschleunigen, wurden zu den wichtigsten Includefiles Units erstellt. Ihr Interfaceteil besteht im wesentlichen aus den Includefiles, während ihr Implementationsteil meist leer ist (bei einigen werden im Init-Teil kleine Initialisierungen vorgenommen).

Wegen des enormen Umfangs dieser Includefiles kann an dieser Stelle nicht jeder dort definierte Bezeichner erwähnt (und schon gar nicht erläutert) werden. Einiges wird in diesem Handbuch noch im Kapitel über Systemprogrammierung erklärt, sonst kann ich Ihnen nur empfehlen, sich die entsprechenden Includefiles einmal anzusehen und auf die diverse Literatur zur AMIGA-Systemprogrammierung verweisen.

8.3.1 Exec1 und Exec

Exec ist der innerste Teil des AMIGA-Betriebssystems. Deshalb werden einige der Exex-Includes auch von anderen Betriebssystemteilen benutzt, während man andere nur braucht, wenn man wirklich direkt Exex programmieren will.

Aus diesem Grund wurden zwei Exec-Units definiert: Zunächst "Exec1", das in seinem Interfaceteil folgende Includefiles enthält:

```
exec/lists.h
exec/nodes.h
exec/ports.h
exec/semaphores.h
exec/tasks.h
```

Das Unit Exec1 wird dann auch von so ziemlich allen anderen Systemunits benutzt. In "Exec" wird dann der ganze Rest definiert, namentlich der Inhalt folgender Include-Dateien:

```
exec/alerts.h
exec/devices.h
exec/execbase.h
exec/errors.h
exec/interrupts.h
exec/io.h
exec/libraries.h
exec/memory.h
exec/resident.h
```

Außerdem wird hier mit "USES" das Unit "Exec1" eingebunden, so daß dann auch die Definitionen dieses Units automatisch zur Verfügung stehen.

8.3.2 GraphTypes und Graphics

Die "Graphics"-Includes sind im wesentlichen in zwei Dateien enthalten: "GraphTypes" und "Graphics". Erstere enthält die wichtigsten Datentypen im Grafik-Bereich, nämlich den Inhalt folgender Dateien:

```
graphics/gfx.unit.h
graphics/clip.h
graphics/copper.h
graphics/view.h
graphics/layers.h
graphics/rastport.h
graphics/text.h
```

Die Datei "gfx.unit.h" ist identisch mit "gfx.h", enthält aber von der dort deklarierten Funktion "RasSize" nur den Kopf (wie es im Interfaceteil von Units sein muß). Der Rumpf dieser Funktion steht im Implementationsteil des Units. Außerdem benutzt dieses Unit das oben beschriebene Unit "Exec1".

Das Unit "Graphics" benutzt "Exec1" und "Graphtypes". Im Interfaceteil steht nur noch die Includedatei "graphics.unit.lib", die sich von "graphics.lib" darin unterscheidet, daß die Prozedur "OpenGfx" fehlt. Statt dessen wird die Graphics-Library im Initialisierungsteil des Units geöffnet. Wenn Sie das Unit "Graphics" benutzen, müssen Sie diese Bibliothek also weder selbst öffnen noch schließen.

Beachten Sie aber, daß diese beiden Units nicht alle Grafik-Includes repräsentieren. Wenn Sie also z. B. mit Gels arbeiten wollen, müssen Sie die Datei "graphics/gels.h" zusätzlich als Includedatei aufrufen.

8.3.3 Intuition

Im Interfaceteil des Units "Intuition" stehen folgende Includedateien:

```
intuition/intuition.h
intuition/screens.h
intuition.lib
```

Außerdem werden die Units "Exec1" und "Graphtypes" benutzt. Der Inhalt der Dateien "intuition/intuitionbase.h" und "intuition/preferences.h" ist also nicht Bestandteil des Intuition-Units. Im Initialisierungsteil des Units wird die Intuition-Library geöffnet.

8.3.4 Custom

Das Unit "Custom" entspricht der Includedatei "hardware/custom.h" und definiert somit die Struktur der Custom-Chip-Register. Außerdem exportiert das Unit eine Variable:

```
VAR cr: Custom ABSOLUTE $DFF000;
```

Somit ist es möglich, über diese Variable die Chipregister direkt anzusprechen. Ein extrem einfallsreiches (aber wenigstens kurzes) Beispiel:

```
Uses Custom;
Var i: Integer;
Begin
  For i:=0 to MaxInt do
    cr.color00:=i
End.
```

Dieses kleine Programm ändert für einige Zeit ständig die Hintergrundfarbe.

8.4 ExecSupport und ExecIO

Zwischen der direkten Programmierung der Hardware und dem Zugriff darauf über Libraries gibt es im AMIGA eine weitere Ebene, Geräte wie z. B. den Drucker oder die Diskettenlaufwerke zu programmieren: Die Devices.

Dabei handelt es sich hauptsächlich um Tasks, die ständig laufen (aber natürlich im Waiting State sind, wenn sie gerade nichts zu tun haben, und somit keine Rechenzeit verbrauchen). Andere Tasks können den Devices durch Messages Aufträge erteilen, die dann - falls möglich - vom jeweiligen Device ausgeführt werden.

Ein Device wird von KICK-PASCAL direkt unterstützt: Das Console-Device, das sich um Texteingaben und -ausgaben in Windows kümmert. Außerdem enthält das AMIGA-Betriebssystem standardmäßig folgende Devices:

- Audio Device: erledigt Soundausgaben. Man kann zwar auch über direkten Zugriff auf die Hardware-Register Töne erzeugen, aber über dieses Device geht das "sauberer" und zum Teil auch einfacher.
- Timer Device: kann praktisch nur eine frei wählbare Zeit lang warten und dann melden, daß die Zeit abgelaufen ist. Man kann es brauchen, wenn man in bestimmten Zeitabständen irgendwelche Aktionen ausführen will.
- Trackdisk Device: Damit können Sie direkt auf die Blöcke und Sektoren von Disketten zugreifen. Wenn Sie eine Festplatte besitzen, verfügt Ihr Betriebssystem außerdem über ein Device, das analog zu "Trackdisk" auf die Harddisk zugreift.
- Input Device: handhabt auf unterster Ebene Benutzereingaben aller Art.
- Keyboard Device: Tastaturtreiber - Gameport Device: befaßt sich mit den beiden "Gameports", also den Maus- und Joystickschnittstellen.
- Narrator Device: ist für die (meist grausam klingende und deshalb nur selten benutzte) Sprachausgabe des AMIGA verantwortlich.
- Serial Device: Treiber für die serielle Schnittstelle. Jeder, der schon einmal die entsprechende Schnittstelle unter MS-DOS programmieren mußte, wird von der Leistungsfähigkeit dieses Devices erfreut sein.
- Parallel Device: Treiber für die parallele Schnittstelle.
- Printer Device: befaßt sich mit dem Drucker und kann z. B. auch Grafiken ausdrucken.
- Clipboard Device: unterstützt das Zwischenspeichern und spätere Wiederverwenden von Daten.

Warum ich Ihnen das alles erzähle? Nun, im AMIGA-Betriebssystem "fehlen" einige Routinen, die (hauptsächlich) die Programmierung von Devices komfortabler machen, nämlich die Exec-Support-Funktionen. Sie sind im "AMIGA ROM Kernel Reference Manual: Exec" als Quelltext enthalten - aber, wie nicht anders zu erwarten, in "C". Diese Funktionen - außer "CreateTask" - stehen Ihnen als KICK-PASCAL-User aber in Form eines Units zur Verfügung: Es heißt "ExecSupport".

Außerdem gibt es noch ein Unit namens "ExecIO", das alles weitere enthält, was man so zum Device-Programmieren braucht.

Aber zunächst zu "ExecSupport". Hier sind im einzelnen folgende Prozeduren und Funktionen implementiert:

PROCEDURE NewList (list: p_List);

Initialisiert die Listenstruktur, auf die der Parameter zeigt.

FUNCTION CreatePort (name: Str; pri: Byte): p_MsgPort;

Erzeugt einen Message-Port unter dem angegebenen Namen und mit der Priorität "pri", und gibt einen Zeiger darauf zurück. Als kleine Besonderheit merkt das Unit ExecSupport sich, welche Ports diese Funktion erzeugt hat, und entfernt diese am Programmende automatisch (mittels eines entsprechenden ExitServers), sofern dies noch nicht geschehen ist.

PROCEDURE DeletePort (port: p_MsgPort);

Ein mit "CreatePort" erzeugter Messageport wird entfernt. Wie bereits oben angedeutet, muß diese Prozedur in Ihrem Programm praktisch nie direkt benutzt werden, da am Programmende automatisch alle erzeugten Messageports gelöscht werden.

FUNCTION CreateStdIO (ioReplyPort: p_MsgPort): Ptr;

Diese Funktion erzeugt eine Standard-IO-Request-Struktur und gibt einem Zeiger darauf zurück. Als Parameter wird ein Zeiger auf einen Messageport benötigt, der z. B. zuvor mit "CreatePort" erzeugt worden sein kann. Wenn die Struktur nicht eingerichtet werden konnte, steigt die Funktion mit einer Fehlermeldung aus. Wie bei "CreatePort" werden auch hier alle eingerichteten IO-Strukturen am Programmende automatisch freigegeben, sofern dies noch nicht geschehen ist.

PROCEDURE DeleteStdIO (ioStdReq: Ptr);

Als Gegenstück zu "CreateStdIO" löscht diese Prozedur eine "normale" IO-Request-Struktur. Allerdings werden diese Strukturen am Programmende auch automatisch gelöscht, so daß Sie "DeleteStdIO" nur selten direkt benutzen werden.

FUNCTION CreateExtIO (ioReplyPort: p_MsgPort; size: Long): Ptr;

Einige Devices benötigen eine um gewisse gerätespezifische Einträge erweiterte IO-Request-Struktur. Mit der Funktion "CreateExtIO" kann man solche Strukturen

beliebiger Größe (Parameter "size" entspricht der Gesamtgröße im Bytes) analog zu "CreateStdIO" erzeugen lassen. Die zusätzlichen Felder werden mit Nullen initialisiert.

Auch diese Funktion steigt bei Mißerfolg mit einer Fehlermeldung aus und merkt sich intern die eingerichteten Strukturen, so daß diese am Programmende automatisch gelöscht werden.

PROCEDURE DeleteExtIO (ioExt: Ptr);

Diese Prozedur löscht eine mit "CreateExtIO" erzeugte Struktur.

PROCEDURE BeginIO (ioReq: Ptr);

Der Aufruf dieser Prozedur entspricht einem Einsprung in eine interne Funktion, die jedes Device besitzt und allgemein eine IO-Operation startet. "BeginIO" ähnelt stark der Exec-Funktion "SendIO", allerdings ist "BeginIO" direkter, d. h. ohne zusätzlichen Exec-"Überbau". Im allgemeinen sollte man "SendIO" benutzen, während "BeginIO" nur in einigen speziellen Fällen sinnvoll benutzt werden kann.

Das Unit "ExecSupport" benutzt übrigens das Unit "Exec1".

Wie bereits erwähnt, gibt es noch ein Unit namens "ExecIO". Es benutzt die Units "Exec1" sowie "ExecSupport" und enthält im Interfaceteil folgende zur Deviceprogrammierung notwendigen Includefiles:

```
'exec/libraries.h'
'exec/devices.h'
'exec/io.h'
```

Außerdem exportiert es noch zwei Prozeduren:

PROCEDURE Open_Device (DevName:Str; Unit:Long; IOReq: Ptr; Flags:Long)

Diese Prozedur entspricht der Exec-Funktion "OpenDevice", bis auf zwei Unterschiede:

1. Konnte das Device nicht geöffnet werden, steigt "Open_Device" mit einer Fehlermeldung aus. Deshalb konnte es auch als Prozedur implementiert werden, während "OpenDevice" eine Function ist, die evtl. eine Fehlernummer zurückgibt.
2. Wieder einmal wird über die geöffneten Devices Buch geführt, so daß sie am Programmende vollautomatisch geschlossen werden und das System immer schön aufgeräumt ist.

PROCEDURE Close_Device (ioRequest: Ptr)

Unterscheidet sich von der Exec-Prozedur "CloseDevice" nur insofern, als daß hier anhand der von "Open_Device" erzeugten Liste geprüft wird, ob das Device auch wirklich geöffnet wurde und daß diese Prozedur verzichtbar ist, da am Programmende alle Devices automatisch geschlossen werden.

8.5 Das Unit Menus

Das Unit "Menus", das im PASCAL geschrieben ist, dient zur einfachen Verwaltung von Pull-Down-Menüs. Es benutzt das Unit "Crt" und kümmert sich nur um Menüs im Crt-Ausgabefenster.

Es liegt im Unterverzeichnis "Windows", muß also mit

```
FROM Windows USES Menus;
```

aufgerufen werden.

Folgende Prozeduren und Funktionen werden zur Verfügung gestellt:

PROCEDURE ClearMenu

Eine bereits erzeugte Menüleiste wird vollständig gelöscht. Übrigens kümmert sich ein Exit-Server des Units darum, daß am Programmende diese Funktion auf jeden Fall aufgerufen wird.

PROCEDURE AddMenu (x: integer; Name: Str; Enable: Boolean)

x: Horizontale Position des Menüs.
Name: Menütitel
Enable: Flag, ob Menü anwählbar sein soll.

Ein neues Menü wird in die Menüleiste aufgenommen. Das zuerst erzeugte Menü hat dann die Nummer 0, das zweite Nummer 1 usw.

PROCEDURE AddItem (y: integer; Flag: Word; Name: Str; Com: Char)

y: Vertikale Position des Menüpunkts (in Bildpunkten).
Flag: Menü-Flags (siehe unten).
Name: Menütitel
Com: Tastaturäquivalent zum Menüpunkt.

Nachdem man mit "AddMenu" ein Menü erzeugt hat, kann man durch wiederholtes Aufrufen von "AddItem" die einzelnen Menüpunkte anhängen. Dabei ist "y" die vertikale Position (die Menüpunkte stehen also stets untereinender), "Name" ist der Textstring und mit "Com" kann man einen Tastatur-Hotkey zu diesem Menüpunkt angeben. Wird hier Space oder chr(0) angegeben, wird kein Tastaturäquivalent gesetzt.

Mit dem Parameter "Flag" kann man einige besondere Eigenschaften des Menüs angeben. Eine "0" entspricht einem ganz normalen Menüpunkt, und im Interfaceteil des Units sind vier weitere mögliche Werte als Parameter definiert:

CheckOff = $001

Das Menü wird bei Anwahl mit einem Haken versehen. Er ist zunächst nicht aktiviert.

CheckOn = $101

Wie "CheckOff", aber der Haken ist von Anfang an aktiv.

ToggleOff = $009

Bei jedem Anwählen des Menüpunkts wird das Häkchen umgeschaltet. Zunächst soll der Menüpunkt nicht angewählt sein.

ToggleOn = $109

Wie "ToggleOff", aber der Menüpunkt ist zunächst abgehakt.

PROCEDURE AddSubItem (y: integer; Flag: Word; Name: Str; Com: Char)

y : Vertikale Position des Untermenüpunkts (in Bildpunkten)
Flag : Menü-Flags (siehe unten)
Name: Untermenütitel
Com: Tastaturäquivalent zum Untermenüpunkt.

Diese Prozedur entspricht genau "AddItem", aber sie hängt an den zuletzt mit "AddItem" erzeugten Menüpunkt einen Untermenüpunkt an.

PROCEDURE MutualExclude (exc: LongInt)

exc: Bitmaske

Für den zuletzt mit "AddItem" oder "AddSubItem" erzeugten Menüpunkt bzw. Untermenüpunkt wird ein "Mutual Exclude" definiert. Das ist eine Bitmaske, die bei

Menüpunkten, die mit einem Haken versehen sind, eine automatische Ausschaltung desselben bewirken. Der Parameter ist eine Bitmaske, bei der die Bits gesetzt werden, wenn der jeweilige Menüpunkt bei Anwahl des zuletzt erzeugten Manüpunktes deaktiviert werden sollen.

PROCEDURE MenuOff

Die Menüleiste wird ausgeschaltet. Im Gegensatz zu "ClearMenu" wird hier die Menüstruktur aber gespeichert und kann mit "MenuOn" später wieder unverändert eingeschaltet werden.

PROCEDURE MenuOn

Eine mit "MenuOff" ausgeschaltete Menüleiste wird wieder aktiviert.

FUNCTION MenuPicked: Boolean

Diese Funktion wird "true", wenn vom Benutzer ein Menüpunkt angewählt wurde.

FUNCTION PickedMenu: integer

Wenn "MenuPicked" wahr ist, liefert die Funktion "PickedMenu" die Nummer des angewählten Menüs.

FUNCTION PickedItem: integer

Analog zu "PickedMenu" liefert "PickedItem" die Nummer des Menüpunkts.

FUNCTION PickedSubItem: integer

Wie "PickedItem" gibt "PickedSubItem" die Nummer des evtl. angewählten Untermenüpunkts zurück, oder den Wert -1, wenn kein Untermenüpunkt angewählt wurde (z. B. wenn der angewählte Menüpunkt gar kein Untermenü hat).

PROCEDURE NextPicked

Mit der linken Maustaste kann man mehrere Menüpunkte gleichzeitig anwählen. Intuition liefert dann aber nur eine Message, und man muß im Prinzip eine Liste von angewählen Menüs durchlaufen. Dazu dient die Prozedur "NextPicked": Damit kann man dem Unit mitteilen, daß man ein einzelnes Menüereignis ausgewertet hat und das nächste bearbeiten möchte. Nach dem Aufruf von "NextItem" kann man mit "MenuPicked" erfragen, ob ein weiteres Menu angewählt wurde.

Beispiel für eine Menüauswertung:

```
WaitForKey;
IF MenuPicked THEN
  BEGIN
    CASE PickedMenu OF
      { Menüs auswerten }
    END
  END;
```

Mit diesem Programmfragment erfahren Sie bei Mehrfachselektionen jeweils nur den ersten Menüpunkt. Korrekter geht es deshalb so:

```
WaitForKey;
WHILE MenuPicked DO
  BEGIN
    CASE PickedMenu OF
      { Menüs auswerten }
    END;
    NextPicked
  END;
```

FUNCTION MenuChecked (menu, item, subitem: integer): Boolean

menu: Menünummer
item: Nummer des Menüpunkts
subitem: Nummer des Untermenüs bzw. -1

Mit dieser Funktion können Sie erfragen, ob bei einem Menüpunkt die Checkmark (das Häkchen) gesetzt ist.

FUNCTION RightButtonPressed: Boolean

Die Funktion gibt an, ob die rechte Maustaste gedrückt wurde, ohne daß ein Menüpunkt angewählt wurde.

FUNCTION MouseClicked: Boolean

Diese Funktion liefert "true", wenn mit der linken Maustaste in das Ausgabefenster geklickt wurde.

FUNCTION IntuiMessageReceived: Boolean

Ist "true", wenn im Crt-Fenster irgendeine Intuition-Message eingetroffen ist. Das kann eine Nachricht über das Schließen des Fensters, Anwahl eines Menüs, Mausklick usw. sein, je nachdem, welche IDCMP-Flags man gesetzt hat.

FUNCTION MouseX: integer

Immer, wenn eine IntuiMessage empfangen wurde, kann man mit dieser Funktion die X-Koordinate der Mausposition erfragen, z. B. nachdem "MouseClicked" erfüllt ist.

FUNCTION MouseY: integer

Wie "MouseX" liefert diese Funktion die entsprechende Y-Koordinate.

Auf Ihrer KICK-PASCAL-Sourcediskette finden Sie ein Beispielprogramm, das die meisten Funktionen des Units "Menus" benutzt.

EINFÜHRUNG IN DIE

AMIGA

SYSTEMPPROGRAMMIERUNG

Der AMIGA hat ohne Zweifel ein sehr leistungsfähiges Betriebssystem. Andererseits ist er gerade dadurch schwieriger zu programmieren als z. B. ein MS-DOS-Rechner.

Einige Beispiele:

- Während man unter eher konventionellen Betriebsystemen einfach etwas auf dem Bildschirm ausgeben kann, muß man beim AMIGA immer irgendwie angeben, in welches Window die Ausgabe erfolgen soll.
- Direkter Zugriff auf die Hardware (z. B. Bildschirmspeicher) ist tabu, denn dabei können mehrere Tasks durcheinander kommen.
- Wer eine Warteschleife programmiert, macht sich bei Anwendern des Programms unbeliebt, denn eine unnötige Schleife "verheizt" Rechenzeit, die vielleicht gerade ein anderer Task brauchen kann.

Dieses Kapitel soll vor allem Anfänger in die Systemprogrammierung unter PASCAL einführen. Vielleicht findet hier aber auch der Fortgeschrittene den einen oder anderen Tip, den er noch nicht kennt.

Es werden keine nennenswerten AMIGA-Kenntnisse vorausgesetzt, allerdings sollten Sie sich mit PASCAL im allgemeinen (und Pointern im besonderen!), den verschiedenen Zahlensystemem (Hex und Binär, Bytes, Worte und Langworte) und logischen Verküpfungen (also AND, OR, XOR...) von Bits etwas auskennen. Es geht jedenfalls ganz einfach los, auch wenn's kompliziert klingt.

SYSTEMPROGRAMMIERUNG

KAPITEL I

CON:, RAW: UND WAS MAN DAMIT ALLES MACHEN KANN

I. CON:, RAW: und was man damit alles machen kann

1.1 Escape-Sequenzen

Wenn Ihr in PASCAL geschriebenes Programm vom CLI gestartet wird, erfolgen Aus- und Eingaben (also Write/WriteLN und Read/ReadLN) in (was sonst?) das CLI-Fenster. Dabei handelt es sich um ein sog. CON:-Window (leitet sich von "Console" ab - was das genau ist, werden Sie später erfahren). Auch das PASCAL-System simuliert, wenn ein Programm von dort gestartet wird, ein solches Fenster. Deshalb lohnt es sich, wenn Sie sich näher mit den Möglichkeiten solcher Fenster vertraut machen.

Die wichtigsten Möglichkeiten zur Programmierung solcher Fenster kennen Sie bereits: die PASCAL-Befehle Write(ln), Read(ln) und Page.

Page löscht ganz einfach den Bildschirm. Über Read(ln) ist auch nicht viel zu sagen: Es werden bei der Standardeingabe "output" stets ganze Zeilen gelesen; eine einfache Tastaturabfrage ist bei "CON:" nicht möglich. Dafür kann man mit Write(Ln) eine ganze Menge machen. Der Schlüssel zu all' diesen Möglichkeiten wurde bereits im Abschnitt über die Stringtypen angedeutet: die Escape-Sequenzen.

Wie Sie vielleicht wissen, ist nicht jedes Zeichen des Datentyps "Char" direkt auf dem Bildschirm darstellbar (nämlich die Zeichen von chr(0) bis chr(31) sowie von chr(128) bis chr(159)). Vielmehr haben einige von ihnen, wenn sie mit Write(Ln) ausgegeben werden, eine besondere Wirkung auf das Ausgabefenster bzw. den Cursor:

chr(8) Entspricht der BackSpace-Taste: der Cursor geht um eine Position nach links. Allerdings wird dabei kein Zeichen gelöscht.

chr(10) "LineFeed": Der Cursor geht eine Zeile nach unten und springt im allgemeinen dabei auch an den Zeilenanfang.

chr(11) Eine Zeile nach oben.

chr(12) Bildschirm löschen (wie "Page").

chr(13) ("Carriage Return" = "Wagenrücklauf" - dieser Name stammt aus der Zeit, als es noch keine Bildschirme gab und Ausgaben normalerweise auf einen Drukker erfolgten) Cursor an Anfang der Zeile setzen.

chr(15) Auf Sonderzeichen umschalten.

chr(14) Schaltet nach chr(15) wieder auf normale Ausgabe zurück.

Dazu ein kleines Demoprogramm:

```
Program CodesDemo(input,output);
Const
  BackSpace = chr(8);
  ClrScr = ''$c;      { entspricht chr($c)=chr(12) oder #12 - siehe
                        Kapitel "Stringtypen". }
```

```
Begin
  write(clrScr,chr(10));
  { Bildschirm löschen und eine Zeile nach unten }
  write('abcx',BackSpace,'def');
  { Bewirkt die Ausgabe von "abcdef", da das "x" durch das Backspace
    mit "d" überschrieben wird. }
  write(' unten'\ll'oben');
  { Gibt erst "unten" aus und springt dann eine Zeile nach oben , wo
    "oben" ausgegeben wird. }
  writeln; { identisch mit: write(chr(10)) }
  writeln;
  writeln('      soft',chr(13),'Himpel');
  { Ergebnis: "Himpelsoft" }
  writeln('Normal',chr(14),' nicht mehr Normal',chr(15))
end.
```

Auf dem Bildschirm steht am Programmende folgendes:

```
   oben
abcdef unten
Himpelsoft Normal!??????:????  <- diese Hieroglyphen sind nicht
                                  tippbar!
```

Der AMIGA kann aber noch viel mehr - mehr, als freie Char-Codes zur Verfügung stehen! Die Lösung (und damit wären wir endlich beim Thema) sind die Escape-Sequenzen.

Eine solche Sequenz besteht (wie der Name schon sagt) aus mehreren Zeichen, deren erstes das Zeichen "Escape" ist. Es hat den ASCII-Code $1b oder dezimal 27. Dieses Zeichen teilt dem Betriebssystem mit, daß die folgenden Zeichen nicht ausgegeben werden, sondern etwas besonderes bewirken sollen. Bei den Sequenzen, die uns hier interessieren, ist das zweite Zeichen stets eine geöffnete eckige Klamme; sie beginnen also mit

```
''\27'['  bzw.  #27'['
```

Dafür kennt das AMIGA-Betriebssystem eine Art Abkürzung: Das Zeichen mit dem Code $9b oder dezimal 155, das auch kurz "CSI" genannt wird (abgeleitet nicht von "BMW 635CSi", sondern von "Control Sequence Introducer").
Gleich zum ersten Beispiel: "CSI-@"

```
Program InsertSpaces(input,output);
  Begin
    write('text1text2',chr(8),chr(8),chr(8),chr(8),chr(8));
    write(chr($9b),'4@')
  end.
```

Die erste "write"-Zeile dürfte Ihnen klar sein: es wird "text1text2" ausgegeben und dann fünfmal mit dem bereits erwähnten BackSpace der Cursor um jeweils eine Position zurückgesetzt, so daß er jetzt auf den zweiten "t" steht. In der zweiten Zeile wird zunächst mit Chr($9b) eine Escapesequenz eingeleitet. Es folgt das Zeichen "4" - eine Art Parameter der Sequenz - und dann das entscheidende Zeichen "@" (Sie können es auf der deutschen Tastatur mit "Alt-2" tippen). Diese Sequenz fügt

Leerzeichen ein - in diesem Fall vier Stück (wenn Sie zwischen "CSI" und "@" keine Dezimalzahl angeben, wird genau ein Leerzeichen eingefügt).

Wenn Sie das Kapitel über die Stringtypen aufmerksam gelesen haben, wissen Sie sicherlich noch, daß Sie denselben Text auch einfacher ausgeben können, indem Sie ihn einschließlich aller Steuerzeichen in einen einzigen String packen:

```
write('textltext2'\8\8\8\8\e'4@')
```

(Zur Erinnerung: "\e" steht für das CSI-Zeichen. Man hätte statt dessen auch "\155" oder "$9b" schreiben können).

Aber es geht noch einfacher: Auch für die Cursorsteuerung in alle Richtungen gibt es Steuersequenzen:

CSI-n-'A'	n Zeilen nach oben
CSI-n-'B'	n Zeilen nach unten
CSI-n-'C'	n Spalten nach rechts
CSI-n-'D'	n Spalten nach links

Unser erstes Beispiel könnte man so verkürzen:

```
write('textltext2'\e'5D'\e'4@')
```

Wieder bedeutet eine weggelassene Zahl "1", so daß z.B. chr($9b),'A' bei der Ausgabe dieselbe Wirkung wie chr(11) hat.

Ein weiteres Beispiel:

```
Program Rechteck(input,output);
 Var x,y: integer;
 Begin
   write('Breite: '); readln(x);
   write('Höhe:   '); readln(y);
   Page;
   write('*'\e, x-2, 'C*'\e, y-1, 'B'\e'D*'\e, x, 'D*');
 End.
```

Die Sequenz sieht ganz schön wild aus, nicht? Sie gibt vier Sterne aus, die die Ecken eines "x" Spalten breiten und "y" Zeilen hohen Rechtecks darstellen. Was geschieht aber im einzelnen? Zuerst wird der Stern oben links ausgegeben. Es folgen ein CSI, die Dezimalzahl "x-2" und ein "C", d. h. der Cursor springt um x-2 Spalten nach rechts, wo dann sofort der zweite Stern ausgegeben wird. Als nächstes geht es mit der Sequenz ESC, y-1, 'B' um y-1 Zeilen nach unten und dann mit CSI-D um eine Position nach links, wo der dritte Stern geschrieben wird. Zu guter Letzt wird noch mit Esc, x, 'D','*' der vierte Stern um x Spalten nach rechts versetzt ausgegeben.

Weitere Sequenzen zur Cursorsteuerung:

CSI, x, 'E'
setzt den Cursor um x Zeilen runter in die erste Spalte. Defaultwert für "x" ist wieder einmal 1.

Beispiel:
```
write(chr($9b),'12E');
```

CSI, x, 'F'
wie oben, aber aufwärts.

Beispiel:
```
write(chr($9b),'F');
```

CSI, y, ';', x, 'H'
Cursor in Zeile y und Spalte x setzen. Während Sie ihn mit den bisher beschriebenen Sequenzen nur um eine Distanz verschieben konnten, können Sie ihn hiermit an eine absolute Position setzen. Die linke obere Ecke des Windows hat übrigens die Koordinaten (1,1).

Beispiel: In gewissen PASCAL-Implementationen unter MS-DOS gibt es die Prozedur "GotoXY". Auf dem AMIGA können Sie das so machen:

```
Procedure GotoXY(x,y:integer);
 Begin
  write(chr($9b),y,';',x,'H')
 End;
```

Weitere nützliche Sequenzen:

CSI, 'J'	Fenster ab Cursor löschen.
CSI, 'K'	Zeile ab Cursor löschen.
CSI, 'L'	Zeile einfügen.
CSI, 'M'	Zeile löschen.
CSI, '0 p'	Cursor unsichtbar machen.
CSI, ' p'	Cursor wieder sichtbar machen (die Spaces in diesen Sequenzen sind wichtig!).

CSI, para1, ';', para2, ';', para3, ..., 'm'
Setzt Textattribute. Es können beliebig viele Parameter angegeben werden, die jeweils Dezimalzahlen sind und folgende Bedeutung haben:
0-7: Stil
0 = normal
1 = fett
3 = schräg ("italic")
4 = unterstrichen
7 = invers
30-37: Vordergrundfarben 0 bis 7
40-47: Hintergrundfarben 0 bis 7
In einem normalen Workbench-Window haben Sie natürlich nur die Farben 0 bis 3 zur Verfügung.

Beispiel zur letzten Sequenz:

```
Program TextSpielereien(output);

 Procedure SetStyle(Stil:integer);
   Begin
    write(chr($9b),Stil,'m')
   End;

 Procedure SetCol(Vfarb,Hfarb:integer);
   Begin
    write(chr($9b),'3',Vfarb,';4',Hfarb,'m')
   End;

Begin
  Page;
  SetStyle(1);
  SetCol(3,2);
  writeln('Fett und Rot auf schwarz');
  SetStyle(4);
  writeln('zusätzlich noch unterstrichen');
  SetCol(2,1);
  SetStyle(3);
  writeln('Schwarz auf weiß - und kursiv');
  SetStyle(0);
  writeln('Wieder normal.')
End.
```

Die im Programm genannten Farben beziehen sich übrigens auf die Standardeinstellungen der Workbench. Wenn Sie in den Preferences andere Farben eingestellt haben, stimmen die Texte natürlich nicht.

Mit diesen Sequenzen können Sie doch schon einiges realisieren. Es gibt noch einige mehr, aber irgendwann muß Schluß sein - schließlich hat das AMIGA-Betriebssystem noch viele andere erwähnenswerte Sachen auf Lager. Zum Beispiel CON:- und RAW:-Fenster.

1.2 CON:-Fenster

Was genau ist "CON:"? Sicher kenne Sie "DF0:" oder "RAM:". CON: ist ebenso ein Gerät, das vom AMIGA-DOS aus verwaltet werden kann (dieses "Gerät" ist natürlich keins, das man anfassen kann, sondern einfach nur ein Programm, das Windows auf dem Workbench-Bildschirm verwaltet).

Mit diesem Gerät können Sie Windows (Fenster) wie Textdateien behandeln. Ein Dateiname sieht dabei so aus:

```
Con:x/y/b/h/name
```

Dabei bedeuten:

x/y:	Position der linken oberen Ecke des Fensters
b:	Fensterbreite
h:	Fensterhöhe
name:	Name des Fensters

Aber gehen wir doch gleich zum ersten Beispiel über:

```
Program Fensterei;
Var f: text;  { prinzipiell ist jeder Filetyp möglich, aber ein
                Textfile ist wohl das einzig sinnvolle. }

Begin
  Reset(f, 'Con:40/20/560/180/Ein Window'); { Fenster öffnen }

  If eof(f) Then error('War nix.');          { Fenster konnte nicht
                                               geöffnet werden. }
  Writeln(f, 'Horrido!');                  { Dieser Text wird in
                                             das Fenster geschrieben. }
  Delay(100);  { 2 Sekunden Pause. }

  Close(f)     { Fenster wieder zumachen. }
End.
```

Dieses Programm öffnet in der Mitte des Workbench-Bildschirms ein 560 Punkte breites und 180 Punkte hohes Fenster mit dem Namen "Ein Window". Beachten Sie bitte, daß eine solche Datei aus unerklärlichen Gründen mit "Reset" geöffnet werden muß - bei "ReWrite" gibt es einen Fehler.

Zu Anfang dieses Kapitels wurde bereits erwähnt, daß ein CLI-Fenster normalerweise auch so ein CON:-Fenster ist und das PASCAL-System- Window sich wie ein solches verhält. Deshalb können Sie auch alle oben beschriebenen Escape-Sequenzen auf ein solches selbstgeöffnetes Fenster anwenden.

1.3 RAW: - wo rohe Tasten sinnlos walten

RAW: ist ein enger Verwandter von CON:. Die Syntax der Dateinamen und die Ausgaben in ein RAW:-Fenster sind genau wie bei CON:. Die Unterschiede liegen bei der Eingabe: Während CON: immer ganze Zeilen liest, einschließlich der Möglichkeit, Eingaben mit Backspace oder Ctrl-X zu löschen, liest RAW: die Zeichen mehr oder weniger direkt von der Tastatur (eher "weniger direkt" - aber dazu später mehr). Ein Beispiel: Sie lesen mit "Readln(datei, i)" eine Integerzahl ein. Der Benutzer tippt im Fenster erst "40", drückt dann aber einmal die BackSpace-Taste und korrigiert zu "42", bevor er die Eingabe mit RETURN beendet. In einem CON:-Window wird nun korrekt die Zahl "42" gelesen. Das RAW:-Fenster sendet dagegen die Zeichenfolge "4"-"0"-BackSpace-"2"-RETURN an das PASCAL-Programm (also die "rohen" Tastencodes - daher der Name "Raw"). Das PASCAL-System interpretiert das BackSpace als Zeilenende (da es kein druckbares Zeichen ist), so daß die Zahl "40" gelesen wird.

Die "2" und das RETURN werden zunächst einmal aufbewahrt und dann beim nächsten Read/ReadLn gelesen. Übrigens sieht der Benutzer im RAW:-Fenster gar nicht, was er tippt (dieser Festertyp hat also kein sog. "Echo").

Sie dürften wohl eingesehen haben, daß es wenig sinnvoll ist, in einem solchen Window mit Read(ln) Zahlen oder Strings einzulesen. Es ist aber ideal geeignet, wenn man Chars einlesen will.

Ein etwas längeres Beispiel:

```
Program TastaturMenue;
Var t: text;
    f: integer;
    c: char;
Begin
  Assign(t, 'RAW:0/0/300/100/Rawdemo');
  Reset(t);
  If eof(t) Then Error('Tilt!');
  f:=1;    { Textfarbe }
  Repeat
    Page(t);
    writeln(t, chr($9b), f+30, 'm');    { auf Farbe i schalten }
    writeln(t, 'Menü:');
    writeln(t, '1 Weiß');
    writeln(t, '2 Schwarz');
    writeln(t, '3 Rot');
    writeln(t, 'Esc = Ende');
    write  (t, 'Ihre Wahl: ');
    read(t,c);
    Case c Of
    '1': f:=1;
    '2': f:=2;
    '3': f:=3
    Else ;    { sonst nichts }
  Until c=chr($1b);    { das ist der Code der ESC-Taste }
  Close(t)
End.
```

Dieses Programm erzeugt in einem Window ein kleines Menü mit vier Punkten: Mit den Tasten 1 bis 3 können Sie sich das Menü in drei verschiedenen Farben ausgeben lassen, mit der ESC-Taste kommen Sie aus dem Programm wieder heraus.
Wenn Sie in dem Programm RAW: durch CON: ersetzen, ändert sich folgendes:

- Das Programm wartet bei der Eingabe immer darauf, daß Sie die RETURN-Taste drücken.
- Dann liest es das Zeichen und beim nächsten Schleifendurchlauf den Code der Return-Taste, was ein Flackern des Menüs bewirkt (das Menü wird nach Lesen des Return-Codes noch einmal neu aufgebaut).
- Sie können auch mehrere Ziffern nacheinander eingeben. Die entsprechenden Menüpunkte werden dann nacheinander ausgeführt, z.B. wird bei Eingabe "123123"-Return die Schleife siebenmal durchlaufen.

Etwas kompliziert wird es übrigens, wenn Sie die Cursor-Tasten abfragen wollen. Wenn eine solche Taste gedrückt wird, ergibt das nicht ein einzelnes Zeichen, sondern eine Steuersequenz, wie wir sie bereits kennen. Zum Beispiel sendet die Taste "Cursor Links" die Zeichenfolge CSI-'D'.

Ich erwähnte oben einmal, daß RAW: nicht wirklich "rohe" Tasten-Codes liefert. Tatsächlich nimmt Ihnen das System bei solchen Windows schon eine ganze Menge Arbeit ab: Es wird dafür gesorgt, daß die Eingabe im gerade aktiven Window erfolgt, besondere Tastaturbelegungen (z.B. deutsch - "QWERTZ") werden berücksichtigt und viele Tasten werden, wenn sie längere Zeit gedrückt bleiben, wiederholt gesendet.

Die Ein- und Ausgabe in RAW:-Windows ähnelt übrigens stark der des Console-Device, das bei den Fenstern zum Einsatz kommt, die ohne den Umweg über das DOS direkt unter Intuition geöffnet werden. Und damit wären wir auch schon beim nächsten Thema...

SYSTEMPROGRAMMIERUNG

KAPITEL II

INTUITION

II. Intuition

"Intuition" heißt der Bereich des AMIGA-Betriebssystems, der für Windows, Screens, Menüs, Gadgets und dergleichen mehr zuständig ist (auch das DOS benutzt, wenn es, wie oben beschrieben, ein CON:- oder RAW:-Window öffnet, die Intuition-Library). Zunächst aber brauchen wir uns um diesen Betriebsystemteil gar nicht direkt zu kümmern, denn KICK-PASCAL stellt die wichtigsten Intuition-Funktionen in vereinfachter Form, nämlich als PASCAL-Prozeduren und Funktionen, zur Verfügung. Also legen wir gleich los und machen ein Fenster auf.

2.1 Fenster öffnen - die zweite

"Schon wieder?", werden Sie jetzt vielleicht stöhnen, "Das hatten wir doch schon!". Stimmt, aber wenn Sie ein Fenster mit Umweg über das DOS als "CON:" oder "RAW:" öffnen, können Sie nur sehr wenige der Möglichkeiten nutzen, die Intuition Ihnen bietet.

Zum Beispiel wird Ihnen aufgefallen sein, daß jene Fenster kein Closegadget haben (für blutige Anfänger: das ist das Ding in der Ecke oben links, auf das Sie immer klicken, wenn Sie das Window schließen wollen). Ferner können Sie in einem derartigen Fenster nicht mit Pull-Down-Menüs oder Gadgets arbeiten usw.

Der Funktion, die Ihnen ein Window und damit das Tor zu den enormen Fähigkeiten von Intuition öffnet, ist Ihnen vielleicht schon im Kapitel "Prozeduren und Funktionen" aufgefallen, und sei es auch nur deshalb, weil sie furchtbar viele Parameter hat: Richtig, sie heißt "Open_Window".

In aller Kürze sieht ein Aufruf dieser Funktion etwa so aus:

```
Handle:=Open_Window(X,Y,B,H,Farb,IDCMP,Flag,Name,Scr,MnB,MnH,MxB,MxH);
```

In aller Ausführlichkeit haben die Parameter folgende Bedeutung:

X, Y: Position des Fensters

B, H: Breite und Höhe

Farb: Farbe des Rahmens und der Leisten. Normalerweise wird man hier 1 (weiß) einsetzen. Sie können Ihre Augen aber auch mit einem roten Fensterrahmen vergewaltigen.
Genaugenommen handelt es sich hier um einen Parameter des Typs "Word", der gleich zwei Farben enthält: das niederwertige Byte die oben erwähnte Rahmenfarbe und das höherwertige Byte die Schrift farbe des Fensternamens. Wenn Sie als Parameter 1=$0001 angeben, ist letztere Farbe 0 (also bei Standardfarben blau). Mit $0302 würden Sie einen schwarzen Rahmen und rote Schrift erhalten.

IDCMP: Dieses kryptische Akronym steht für (tief Luft holen) "Intuition Direct Communication Message Port". Wahrscheinlich sagt das Ihnen nicht allzu viel - hier also die Erklärung: Der Benutzer kann mit Hilfe der Maus, der Tastatur o. ä. vieles mit dem Fenster machen, etwa das Schließgadget anklicken, die Größe verändern... Sie haben als Programmierer nun die Möglichkeit, dies alles zu erfahren, und zwar dadurch, daß Sie bei einem derartigen Ereignis von Intuition eine "Message" erhalten. Die Möglichkeiten sind so vielfältig, daß vieles davon Sie in der Praxis gar nicht interessieren wird.
Der Parameter "IDCMP" des Open_Window-Befehls ist ein Langwort, in dem jedes der Bits 0 bis 20 für ein bestimmtes Ereignis steht. Die wichtigsten sind:

Bit	Wert	Name	Bedeutung
1	$00002	NEWSIZE	Verändern der Fenstergröße.
2	$00004	REFRESHWINDOW	Fensterinhalt muß 'refresht' werden.
3	$00008	MOUSEBUTTONS	Drücken einer Maustaste.
4	$00010	MOUSEMOVE	Bewegen der Maus.
5	$00020	GADGETDOWN	Anklicken eines Gadgets.
6	$00040	GADGETUP	Ähnlich wie GADGETDOWN.
8	$00100	MENUPICK	Anwahl eines Pull-Down-Menüs.
9	$00200	_CLOSEWINDOW	Anklicken des Schließ-Gadgets.
10	$00400	RAWKEY	Tastendruck
14	$04000	NEWPREFS	Ändern der Preferences.
15	$08000	DISKINSERTED	Einlegen einer Diskette.
16	$10000	DISKREMOVED	Herausnehmen einer Diskette.
18	$40000	ACTIVEWINDOW	Aktivieren des Fensters.
19	$80000	INACTIVEWINDOW	Deaktivieren des Fensters.

Wenn Sie mehrere Ereignisse abfragen wollen, sind die Werte mit OR (logisches oder) zu verknüpfen, was in diesem Fall (aber nur ausnahmsweise!) dasselbe wie eine Addition ist.
Um beispielsweise das Schließen eines Fensters, einen Tastendruck und eine Größenänderung mitgeteilt zu bekommen, ist als IDCMP-Maske NEWSIZE or _CLOSEWINDOW or RAWKEY = $602 zu wählen.

Flag: Diese Window-Flags bestimmen verschiedene Eigenschaften des Fensters. Hier ist eine kurze Übersicht über die wichtigsten von ihnen:

Bit	Wert	Name	Bedeutung
0	$0001	WINDOWSIZING	Fenstergröße veränderbar (Sizegadget unten rechts erwünscht).
1	$0002	WINDOWDRAG	Window erhält Ziehbalken und ist dadurch verschiebbar.
2	$0004	WINDOWDEPTH	Fensterüberlagerung möglich.
3	$0008	WINDOWCLOSE	Fenster erhält Schließ-Gadget.
6	$0040	SIMPLE_REFRESH	Neuaufbau des Fensterinhalts erfolgt "manuell".
7	$0080	SUPER_BITMAP	Es ist eine Bitmap zu reservieren, in der der Fensterinhalt gespeichert wird.
8	$0100	BACKDROP	Fenster liegt immer hinten.
10	$0400	GIMMEZEROZERO	Besonderer Modus
11	$0800	BORDERLESS	Randloses Fenster
12	$1000	ACTIVATE	Fenster wird beim Öffnen aktiviert.

Hier dürften wohl noch einige zusätzliche Informationen nötig sein: Die Bits 0 bis 3 geben an, welche Gadgets das Fenster haben soll. Mit den Bits 6 und 7 wählen Sie die Refresh-Art. Wie Ihnen zweifellos schon aufgefallen ist, können Windows übereinander liegen, so daß sie teilweise vorübergehend verdeckt sein können. Wenn nun das zuoberst liegende Fenster entfernt und der bisher verdeckte Teil wieder sichtbar wird, muß dessen Inhalt wieder aufgebaut werden - das nennt man Refresh. Es gibt drei Refresh-Arten:

- Simple Refresh: Man sollte besser "No Refesh" sagen, denn Intuition begnügt sich hier damit, Ihnen eine Message zukommen zu lassen, daß ein Refresh nötig wäre, der Rest ist dann die Sache Ihres Programms.
- Smart Refresh: Hier bewahrt Intuition die Teile auf, die verdeckt werden, und setzt sie wieder ein, sobald sie auf dem Bildschirm sichtbar werden. Auch Ausgaben in die verdeckten Teile sind möglich. Diese Art von Refresh ist wohl die beliebteste.
- Super Bitmap: Es wird eine Bitmap (ein Bildschirmspeicher) reserviert, die so groß ist, wie das Fenster maximal werden kann. Jede Ausgabe, die in das Fenster erfolgt, wird erst intern auf diesem Pseudo-Bildschirm ausgeführt, dann wird das Ergebnis auf den wirklichen Bildschirm übertragen.
  Dieses Verfahren frißt enorm viel Speicher, hat aber gegenüber dem "Smart Refresh" den Vorteil, daß auch die Fensterbereiche, die

durch Verkleinern des Fensters vorübergehend unsichtbar werden, in der Bitmap gespeichert bleiben und bei einem späteren Vergrößern wieder auf dem Bildschirm erscheinen.

Für die erste Möglichkeit müssen Sie das Windowflag 6 setzen, für das letzte das siebte Bit. Wenn Sie "Smart Refresh" bevorzugen, ist kein Bit zu setzen.
Mit dem Bit 8 erhalten Sie ein sogenanntes Backdrop-Window. Ein solches Fenster liegt immer ganz hinten, auch dann, wenn ein anderes mit dem Back-Gadget nach hinten geklickt wird. Das Hintergrundfenster der Workbench, in dem die Diskettenicons ausgegeben werden, ist ein solches Backdrop-Fenster.
Normalerweise sind der Ziehbalken, die Gadgets und der Rahmen eines Fensters Bestandteil der Windowfläche. Dann müssen Sie aber aufpassen, daß Sie diese Bereiche nicht mit irgendwelchen Ausgaben überschreiben oder -malen. Wenn Sie aber das GIMMEZEROZERO-Bit setzen, sind in dem Fenster die Leisten und die eigentliche Fensterfläche getrennt. In diesem Fall liegt der Nullpunkt des fensterinternen Koordinatensystems (das uns später noch begegnen wird) unmittelbar unter dem Ziehbalken (falls vorhanden) und rechts von linken Rahmen.
Ein BORDERLESS-Window (Bit 11 gesetzt) hat keinen Rahmen. Auch hierfür ist das Workbench-Hintergrundfenster ein Beispiel. Es ist durch das Fehlen eines Rahmens so unauffällig, daß man es eigentlich gar nicht zur Kenntnis nimmt...
Zu guter letzt noch ein Bit, das eigentlich immer gesetzt sein sollte: ACTIVATE. Es bewirkt, das Ihr Fenster, wenn es geöffnet wird, sofort aktiv ist.

Name: Zeiger auf den Fensternamen (Datentyp "Str")

Scr: Zeiger auf den Screen, in dem das Window erscheinen soll. Wenn es auf dem Workbench-Screen geöffnet werden soll, ist hier NIL einzusetzen.

MinB, MinH, MaxB, MaxH:
Minimale und maximale Breite bzw. Höhe des Fensters. Es ist im allgemeinen nicht sinnvoll, zuzulassen, daß ein Fenster (vorausgesetzt, es hat ein Größenveränderungsgadget) beliebig verkleinert werden kann (beim Vergrößern ist ja durch die Größe der Screens eine Grenze gesetzt). Deshalb können Sie hier für Breite und Höhe Ihres Fensters Minimal- und Maximal-Werte angeben.

Sie haben also jede Menge Gestaltungsmöglichkeiten für Ihr Fenster. Die Funktion "Open_Window" gibt Ihnen die sog. Windowhandle zurück, die Sie aufbewahren müssen, denn über diese Handle können Sie Ihr Fenster so ansprechen, daß Intuition weiß, welches gemeint ist.

Zum Beispiel brauchen Sie die Handle, wenn Sie das Fenster schließen wollen. Der Befehl dazu heißt "Close_Window(w)" und bekommt als einzigen Parameter eben jene Handle übergeben.

Ein Beispielprogramm für "Open_Window" und "Close_Window":

```
Program IntuitionWindow;
 Var
   wh: Ptr;
 Begin
   wh:=Open_Window(0,0,      { Position }
                   640,200,  { Größe }
                   $0201,    { schwarze Schrift, weißer Rahmem }
                   $0200,    { IDCMP: nur _CLOSEWINDOW }
                   $100f,    { Fester aktiv und alle Gadgets }
                   'Intuitionwindow',    { Name }
                   Nil,      { auf Workbench-Screen }
                   200,40,   { Mindestgröße }
                   640,256);{ maximale Größe }
   Delay(3*50);              { 3 Sekunden warten }
   Close_Window(wh)          { Fenster schließen }
 End.
```

Dieses Programm öffnet ein Fenster und schließt es nach 3 Sekunden wieder. Wenn Sie übrigens den Delay-Befehl durch "Signal:=Wait(-1)" ersetzen (natürlich müssen Sie vorher eine Long-Variable "Signal" deklarieren), wartet es so lange, bis das Close-Gadget angeklickt wird. Warum das so ist, werden Sie im Kapitel über "Messages" erfahren, es hängt aber, wie Sie sich denken können, damit zusammen, daß als IDCMP-Parameter "_CLOSEWINDOW" gewählt wurde.

Falls Sie mit KICK-PASCAL auf einem eigenen Screen arbeiten, müssen Sie bei obigem und allen anderen Programmen, die Windows auf der Workbench öffnen, vor dem Programmstart jeweils den Workbench-Screen nach vorn bringen, denn sonst können Sie ja nichts sehen.

2.2 Screens

Einen Screen öffnet man im Prinzip genau so wie ein Fenster, nur daß man eine andere Funktion dafür verwendet. Sie heißt logischerweise "Open_Screen" und hat folgende Syntax:

```
Handle:=Open_Screen(X,Y,B,H,T,HFarb,VFarb,Mode,Name);
```

Die Parameter:

X, Y: Position des Screens

B, H: Breite und Höhe

T: "Tiefe": Anzahl der Bitplanes

HFarb: Hintergrundfarbe

VFarb: Vordergrundfarbe

Mode: Dieses Wort enthält mehrere Flags, die das Aussehen des Screens näher festlegen. Ihre Namen und Bedeutungen (Auswahl):

Bit	Wert	Name	Bedeutung
2	$0004	INTERLACE	Schaltet den (bei Augenärzten) beliebten Interlace-modus ein.
7	$0080	EXTRA_HALFBRITE	Extra-Halfbrite-Modus (64 Farben)
10	$0400	DBLPF	Trennt Rahmen vom eigentlichen Screen.
11	$0800	HOLDNMODIFY	Bit für HAM-Modus
14	$4000	SPRITES	Ermöglicht die Benutzung von Sprites.
15	$8000	HIRES	Hires-Modus (640 Punkte breit)

Name: Titel des Screens

Die Funktion gibt die sog. Screenhandle zurück, die in Wirklichkeit ein Zeiger auf eine Struktur ist. Diesen Zeiger müssen Sie als Parameter im Aufruf von "Open_Window" angeben, wenn Sie auf Ihrem Screen ein Fenster öffnen wollen. Desweiteren dient er als Parameter für die Close_Screen-Funktion, die einen Screen schließt.

Das obligatorische Demoprogramm:

```
Program Screendemo;
Var
 Scr: Ptr;
 Win: Array[0..63] of Ptr;
 i: integer;
Begin
 Scr:=Open_Screen(0,0,              { Position: normal }
                  320,256,          { Größe }
                  6,                { 6 Planes = 64 Farben }
                  3,1,              { Titel rot auf weiß }
                  $0080,            { Extra-Halfbrite-Modus }
                  'Je bunter, desto besser'); { Name }
 For i:=0 to 63 Do
  Win[i]:=Open_Window((i mod 8)*40, 10+(i div 8)*30, 40, 30,
                      i+$100*(63-i), 0, 50, 'Hi!', Scr,
                      50,50,100,100);
 Delay(2*50);                       { Fünf Sekunden Pause }
 For i:=0 to 63 Do Close_Window(Win[i]);
 Close_Screen(Scr)
End.
```

Dieses Programm öffnet einen LoRes-Pal-Screen (320*256 Punkte) im 64-farbigen Extra-Halfbrite-Modus und darauf nicht weniger als 64 Fenster, die jeweils verschiedenfarbige Rahmen und Titel haben.

Wenn Sie bereits wissen, was eine "BitPlane" ist, können Sie diesen Abschnitt überlesen; falls nicht:

Der AMIGA kann 4096 Farben darstellen - aber (normalerweise) nicht alle gleichzeitig, sondern immer nur eine Auswahl von Farben. Die einfachste Art der Bildschirmdarstellung ist die Benutzung einer Vordergrund- und einer Hintergrundfarbe. Dann gibt es für den Screen einen Speicherbereich, in dem jedem Bildschirmpunkt ein Bit entspricht: ist es gesetzt, hat der Punkt die Vordergrund-, sonst die Hintergrundfarbe (kleines Rechenexempel: eine LoRes-Grafik mit 320 Punkten Breite und 200 Punkten Höhe braucht man 320*200=64000 Bits, also - 8 Bits pro Byte - 8000 Bytes). Ein solcher Speicher heißt Bitplane.

Nun sind zwei Farben nicht gerade viel. Die Lösung: Man nimmt mehrere Bitplanes. Dann stehen für jeden Bildpunkt mehrere Bits zur Verfügung.

In unserem Beispielprogramm gibt es für jeden Punkt sechs Planes, man kann also mit den sechs Bits 2^6 = 64 Farben codieren. Leider sind der Hardware des AMIGA gewisse Grenzen gesetzt (wer hätte das gedacht?). Im LoRes-Modus (320 Punkte breit) kann der AMIGA bis zu sechs Bitplanes pro Screen verwalten, im HiRes-Modus (640 Dots breit) nur vier - und das auch nur mit Krampf, denn bei vier HiRes-Bitplanes wird der Rechner langsamer, weil er so viel mit der Grafik zu tun hat.

Und noch eine Einschränkung gibt es: der Grafikchip kann eigentlich nur 32 Farben darstellen. Deshalb gibt es den "Extra-Halfbrite-Modus", den auch unser Programm "ScreenDemo" benutzt: Dieser Grafikmodus benötigt sechs Bitplanes (ist also nur in LoRes möglich). Von den sechs Bits, die dann auf jeden Bildpunkt entfallen, codieren fünf die Farbnummer (denn 2^5 = 32), während das sechste, wenn es gesetzt ist, eine Halbierung der Helligkeit bewirkt. Das können Sie auch beim Programm "ScreenDemo" sehen: Dort sind die Windows in der oberen Bildschirmhälfte (mit den Rahmenfarben 0 bis 31) heller als die ansonsten gleichfarbigen der unteren Hälfte, die die Farbnummern von 32 bis 63 haben.

2.3 Die Intuition.library

Bisher haben Sie Intuition nur indirekt programmiert: mit einen Umweg über KICK-PASCAL-Prozeduren. Aber natürlich können Sie auch direkt auf die Intuition.library zugreifen. Zu diesem Zweck benötigen Sie eine Includedatei namens "intuition.lib". Also muß in Ihrem Programm irgendwo zwischen dem Kopf "Program xxx(...);" und dem Beginn des Hauptprogramms folgende Compiler-directive stehen:

```
{$incl 'intuition.lib'}
```

Natürlich ist vorher noch der Pfad festzulegen, unter dem der Compiler diese und andere Includefiles suchen soll, und zwar entweder durch eine weitere Compiler-Directive wie

```
{$path 'pascal:include/' }
```

oder mit dem Menü "Suchpfade".

Die Datei lädt noch zwei weitere Dateien ein, nämlich "intuition/intuition.h" und "intuition/intuitionbase.h", welche selbst wieder mehrere kleinere Dateien aufrufen, so daß hier eine ansehliche Schachtelung zustande kommt (die momentane Schachtelungstiefe sehen Sie während des Compilierens an der Anzahl der "i" hinter der Zeilennummer). Während die anderen Dateien verschiedene Konstanten und Datentypen enthalten, stehen in "intuition.lib" selbst die Deklarationen der Prozeduren und Funktionen der Intuition-Library. Außerdem wird hier die Variable deklariert, die die Basisadresse der Bibliothek erhalten soll. Sie heißt "IntBase" und ist vom Typ "Ptr".

Am Anfang Ihres Programms (jedenfalls irgendwo vor dem ersten Aufruf einer Intuition-Bibliotheksfunktion) müssen Sie der Variablen "IntBase" den entsprechenden Wert zuweisen, und zwar mit folgendem Prozeduraufruf:

```
OpenLib(IntBase, 'intuition.library', 0);
```

Danach zeigt "IntBase" tatsächlich auf die Basisadresse der Bibliothek. Beachten Sie aber bitte, daß Sie den Bibliotheksnamen klein schreiben müssen. Nun steht Ihnen die gesamte Bibliothek zur Verfügung.

Übrigens hätten Sie auch das Unit "Intuition" via "Uses" einbinden können. Das hätte die Vorteile gehabt, daß das Compilieren schneller geht und daß man die Library nicht selbst öffnen muß, da das bereits im Initialisierungsteil des Units geschieht. Wir wollen in diesem Handbuch aber generell Includes verwenden, da Sie dann einfacher nachsehen können, was Sie da eigentlich einbinden.

In den Includefiles werden auch die Datentypen "Window" und "Screen" definiert (beide sind Records). Die Pointer, die die Funktionen "Open_Window" und "Open_Screen" zurückgeben, sind Zeiger auf solche Strukturen. So können Sie z. B. immer Position und Größe mit den Feldern "LeftEdge" (X-Koordinate), "TopEdge" (Y-Koordinate), "Width" (Breite) und "Height" (Höhe) der Window-Struktur erfragen.

Beispiel:

```
Program WindowStructDemo;
{$inc1"intuition.lib" }
Const
  x=80;
  y=40;
  b=200;
  h=100;
Var
  Win:  Window;
Begin
  Win:=Open_Window(x,y,b,h,1,0,$1007,'Touch me!',Nil,0,0,640,256);
    { Fenster wie immer öffnen }
  Repeat
    { Eine Warteschleife - miserabler Programmierstil! }
  Until (Win^.LeftEdge <> x) or (Win^.TopEdge <> y)
    or (Win^.Width <> b) or (Win^.Height <> h);
  Close_Window(Win)
End.
```

Dieses Programm öffnet ein Fenster und wartet dann, bis das Fenster verschoben oder seine Größe verändert wird.

Neben den hier verwendetet hat die Window-Struktur noch einige andere nützliche Einträge. Ich werde sie (und auch die Einträge der Screen-Struktur) Ihnen jeweils erläutern, wenn wir sie brauchen. Sie können sich die Strukturen aber auch live und in Action ansehen:

- Starten Sie das Demoprogramm "Mon.p".
- Wählen Sie den Menüpunkt "Address/IntBase". Jetzt wird auf dem Bildschirm der Inhalt der sog. IntuitionBase-Struktur gezeigt (auf die übrigens der Pointer "IntBase" nach dem oben erläuterten "OpenLib"-Aufruf zeigt).
- Drücken Sie nun die "Cursor Runter"-Taste, bis der Cursor auf dem mit "FirstScreen" bezeichneten Feld steht. Dort steht als Hexadezimalzahl die Speicheradresse (was nichts anderes als ein Pointer ist) der ersten Screen-Struktur.
- Wenn Sie jetzt die Taste "J" drücken, wird jene erste Screenstruktur mit dem Inhalt aller ihrer Felder ausgegeben. Das erste Feld ist ein Zeiger auf die nächste Screenstruktur (in PASCAL würde man sagen: es ist eine mit Pointern einfach verkettete lineare Liste), das zweite Feld enthält einen Zeiger auf das erste Window dieses Screens. Sie können nun durch diese Strukturen "blättern", indem Sie jeweils den Cursor auf einen Zeiger setzen und dann "J" tippen.

Nach diesem Exkurs in die Abgründe des AMIGA-Speichers aber zurück zum Thema. Was fällt Ihnen an den bisher erzeugten Fenstern auf? Richtig, sie sehen allesamt reichlich leer aus. Deshalb wollen wir im nächsten Abschnitt etwas in ein solches Fenster ausgeben.

2.4 Texte, Images und Border

Die bequemste Methode, Texte in ein Fenster auszugeben, ist die Verwendung des Console.devices, zumal KICK-PASCAL dies mit leistungsfähigen PASCAL-Befehlen unterstützt. Hin und wieder ist es aber unvermeidlich, zur Textausgabe Intuition zu benutzen, und darum soll dies zunächst unser Thema sein.

Die Prozedur, die uns die Intuition.library zur Textausgabe zur Verfügung stellt, heißt

```
PrintIText(Rast,Itext,X,Y)
```

und hat folgende vier Parameter:

Rast	ist ein Zeiger auf den "RastPort" des Windows. Diese Struktur des AMIGA-Betriebssystems braucht uns aber nur so weit zu interessieren, daß wir wissen müssen, woher wir den Zeiger darauf bekommen: er ist in der Window-Struktur in Form des Feldes "RPort" enthalten.
IText	ist ein Zeiger auf eine Variable des Typs "IntuiText".
X, Y	Koordinaten der Windowposition

Der Datentyp "IntuiText" wird in "intuition/intuition.h" wie folgt definiert:

```
IntuiText = Record
   FrontPen, BackPen, DrawMode: Byte;
   LeftEdge, TopEdge: integer;
   ITextFont: ^TextAttr;
   IText: Str;
   NextText: ^IntuiText
End;
```

Wir brauchen also eine Variable dieses Typs und müssen sie initialisieren. Die Felder haben im einzelnen folgende Bedeutung:

FrontPen, BackPen:	Vorder- und Hintergrundfarbe des Texts
DrawMode:	Textmodus
LeftEdge, TopEdge:	Koordinaten der Textposition
ITextFont:	Zeiger, der die Möglichkeit eröffnet, andere Zeichensätze zu benutzen (ist auf "Nil" zu setzen, wenn einem der normale Font reicht).
IText:	Der eigentliche String
NextText:	Zeiger, mit dem man mehrere IntuiText-Strukturen verketten kann, so daß man mehrere Texte auf einen Schlag ausgeben lassen kann.

Das klingt jetzt alles wahnsinnig kompliziert, also ist es Zeit, daß ein Beispielprogramm kommt.

Hier ist es:

```
Program IntuiTextDemo;
 {$incl"intuition.lib" }
 Var
  Win: ^Window;
  t1,t2,t3: IntuiText;
Begin
  OpenLib(IntBase,'intuition.library',0); { ist hier nötig, da ja die
      Funktion "PrintIText" aus der intuition.library benutzt wird }
  Win:=Open_Window(0,0,640,200,1,0,$1007,'Textausgabe',Nil,100,100,640,200);
     { wie fast schon gewohnt ein Fenster öffnen }
  t1:=IntuiText(1,0,1,0,0,Nil,'Dies ist der erste Text.',Nil);
     { Das Record "t1" wird hier initialisiert. Falls Sie sich über
       die Syntax wundern, lesen Sie bitte im Kapitel "Literale für
       Arrays und Records" nach. }
  PrintIText(Win^.RPort,   { Zeiger auf den Rastport unseres Windows }
             t1,           { Zeiger auf t1 }
             10,20);       { Position }

  t2:=IntuiText(3,0,1,0, 0,Nil,'Zeile 2',t3);
  t3:=IntuiText(3,0,4,0,10,Nil,'Zeile 3',Nil);

  PrintIText(Win^.RPort, t2, 10, 40); { "t2" enthält einen Zeiger auf
                "t3", so daß hier beide Texte ausgegeben werden. }

  Delay(100);
  CloseWindow(Win)
End.
```

Dazu noch einige Anmerkungen:

- Der Modus "0" bewirkt eine reine Textausgabe. Wenn aber als Modus 1 angegeben wird, wird da, wo der Text geschrieben wird, auch der Hintergrund mit der angegebenen Hintergrundfarbe ausgefüllt. Der Mode 4 bewirkt eine invertierte Ausgabe.
- In den Strings können Sie unter Intuition KEINE Escape-Sequenzen verwenden.
- Hier wird die Position, an der die Textausgabe erfolgen soll, immer gleich zweimal angegeben: in der Struktur (Felder "LeftEdge" und "TopEdge") und als Parameter des PrintIText-Befehls. Die beiden Koordinatenpaare werden dabei addiert. Besonders gut sehen Sie das bei der gleichzeitigen Ausgabe von "t2" und "t3": Der gesamte Text wird an der Stelle (10,40) geschrieben, wobei "t2" direkt an dieser Stelle erscheint (Koordinaten (0,0)), während "t3" um zehn Bildpunkte nach unten verschoben wird (Koordinaten (0,10), was als Summe (10,50) ergibt, wenn man das erste Paar dazu addiert).

Wenn Sie ein wenig damit herumexperimentieren, werden Sie feststellen, daß diese Art der Textausgabe zwar etwas aufwendig, aber keineswegs schwierig ist.

Mit einem "Border" kann man eine Folge von zusammenhängenden Linien zeichnen lassen, die unter anderem auch einen Rahmen bilden können (wie die Bezeichnung "Border" nahelegt), aber auch jede andere Form haben können. Solche Border werden unter Intuition ganz ähnlich wie Texte ausgegeben, nur heißt die Prozedur "DrawBorder(Rast,Bord,X,Y)", wobei die Parameter dieselbe Bedeutung haben wie bei "PrintIText", nur daß "Bord" nicht auf eine IntuiText-, sondern auf eine Border-Struktur zeigen muß. Der Datentyp Border ist ebenfalls in der Datei "intuition/intuition.h" definiert, und zwar so:

```
Border = Record
  LeftEdge, TopEdge: integer; { Position }
  FrontPen: Byte;             { Rahmenfarbe }
  BackPen: Byte;              { Hintergrundfarbe (bedeutungslos) }
  DrawMode: Byte;             { Zeichenmodus 0=normal,
                                2=invertierend }
  Count: Byte;                { Anzahl der Eckpunkte }
  XY: Ptr;                    { Zeiger auf die Koordinatentabelle }
  NextBorder: ^Border
End;
```

Außer "Count" und "XY" kennen Sie schon alle Felder von der (ähnlich aufgebauten) "IntuiText"-Struktur. Ein Border ist zunächst eine Folge von Punkten, die z. B. die Eckpunkte eines Rechtecks darstellen. Die Koordinaten dieser Punkte sind in einem "Array[...] of integer" abzulegen, und zwar immer abwechselnd eine X- und eine Y-Koordinate. Auf dieses Feld zeigt "XY", während "Count" angibt, wie viele Koordinaten-PAARE (!) es sind.

Am besten betrachten wir gleich ein Beispiel. Im folgenden Programm zeichnet die Prozedur "Recht" im Window mit dem Rastport "Rast" an der Position (X,Y) ein rotes Rechteck der Breite B und Höhe H:

```
Program BorderDemo;
 {$incl"intuition.lib"}
 Var Win: ^Window;
     i  : integer;

Procedure Recht(Rast:p_RastPort; X,Y,B,H: integer);
   Var Feld: Array[0..9] of integer;
       Bor : Border;
   Begin
     { Koordinaten initialisieren: }
     Feld[0]:=0;      Feld[1]:=0;    { Punkt links oben    }
     Feld[2]:=B;      Feld[3]:=0;          { rechts oben   }
     Feld[4]:=B;      Feld[5]:=H;          { rechts unten  }
     Feld[6]:=0;      Feld[7]:=H;          { links unten   }
     Feld[8]:=0;      Feld[9]:=0;    { nochmal links oben  }
     { Boderstruktur initialisieren: }
     Bor:=Border(0,0,    { keine Verschiebung }
                 3,0,    { Farbe: rot }
                 0,      { Modus: normal }
                 5,      { fünf Paare für vier Linien }
                 ^Feld,  { Zeiger auf Koordinaten }
                 Nil);   { kein weiteres Border }
     DrawBorder(Rast, ^Bor, X, Y)
   End;

Begin
   OpenLib(IntBase,'intuition.library',0);  { nie vergessen! }
   Win:=OpenWindow(0,0,640,200,1,0,$1007,'Rahmem',Nil,100,100,640,200);
   For i:=1 to 10 Do    { zehn Rechtecke }
     Recht(Win^.RPort, 25+12*i, 15+5*i, 50+10*i, 15+10*i);
   Delay(8*50);      { acht Sekunden zum Bewundern }
   CloseWindow(Win)
End.
```

Zum Programm bliebe wohl nur zu sagen, daß man die Initialisierung des Arrays auch kürzer lösen könnte (siehe Kapitel "Literale für Arrays und Records"), und daß man für ein Rechteck fünf Punkte braucht, weil DrawBorder nicht automatisch geschlossene Linien zeichnet, so daß man den ersten Punkt am Ende noch einmal angeben muß.

Als drittes Intuition-Standard-Ausgabeelement wären da noch die Images, also Bilder. Das Verfahren ist wieder ähnlich: Es müssen eine Struktur (die in diesem Fall "Image" heißt) initialisiert und ein Pointer darauf nebst RastPort und Koordinaten an eine Intuition-Prozedur übergeben werden. Die Prozedur heißt "DrawImage(Rast, Imageptr, X, Y)" und der Datentyp "Image" ist (wieder in "intuition/intuition.h") folgendermaßen definiert:

```
Image=Record
  LeftEdge, TopEdge: integer;      { Position }
  Width, Height: integer;          { Breite und Höhe des Bildes }
  Depth: integer;                  { Anzahl der Planes }
  ImageData: Ptr;                  { Zeiger auf Bilddaten }
  PlanePick, PlaneOnOff: Byte;     { Farbinformationen }
  NextImage: ^Image                { Zeiger auf weitere Image-Struktur }
End;
```

Nach der Positionsangabe folgen die Maße des Images in Bildpunkten. Das fünfte Feld gibt die Anzahl der Ebenen an (mit einer Ebene kann man nur eine Vordergrund- und eine Hintergrundfarbe benutzen). "ImageData" zeigt auf die eigentlichen Bilddaten: ein "Array[...] of Word", in dem die Bildpunkt zeilenweise Bit für Bit stehen.

Die beiden nächsten Bytes geben im Prinzip die Farben des Images an: In "PlanePick" steht jedes Bit für eine Bitplane und gibt an, ob in dieser Plane des Screens das Image ausgegeben werden soll. So bekommen Sie bei einer Imageplane mit dem "PlanePick=1" ein weißes und mit "2" ein blaues Bild. "PlaneOnOff" gibt an, was in den einzelnen Planes mit den Bits geschehen soll, die in der Imageplane nicht gesetzt sind: Ist das entsprechende Bit in "PlaneOnOff" gesetzt, werden diese Punkte in der Bildschirmplane gesetzt (klingt kompliziert, aber Sie können im folgenden Beispielprogramm ja etwas herumprobieren).

Die Sache hat noch einen Schönheitsfehler: Die Bilddaten, auf die "ImageData" zeigt, müssen im Chip-Memory liegen. Das ist der Speicherbereich, auf den der Grafikcoprozessor zugreifen kann, denn mit seiner Hilfe (und damit mit der berühmten AMIGA-Geschwindigkeit) wird das Bild ausgegeben. Also können Sie das Array nicht als gewöhnliche Variable deklarieren, denn dann wissen Sie ja nie, in welchem Speicherbereich sie zu liegen kommt. Deshalb greifen wir zur Funktion Alloc_Mem, wandeln den LongInt-Wert, den sie zurückgibt, in einen Zeiger um und weisen diesen an einen POINTER auf ein Array zu.

In gewohnter Weise gehen wir nun gleich zum Programmieren über. Es soll ein Bild ausgegeben werden, das 48 Punkte (also drei Worte) breit und 12 Punkte hoch ist. Da es bei den Bilddaten auf die Bits ankommt, gibt man sie am besten als Binärzahlen an.

```
Program ImageDemo;
{$incl"intuition.lib"}
Type
  Plane = Array[1..36] of Word;        { 3*12 = 36 }
Var
  Win: ^Window;
  Img: Image;
  Dat: ^Plane;
Begin
  OpenLib(IntBase,'intuition.library',0);
Win:=Open_Window(160,50,100,50,1,0,$1007,'Image',Nil,100,50,640,200);
  { Speicher für Bilddaten reservieren: }
Dat:=Ptr(Alloc_Mem(SizeOf(Plane), 2));    { 2 = "MEMF_CHIP" }
  { Bild initialisieren: }
Dat^:=Plane(%0001111111111111, %1111111111111111, %1111111111111000,
            %0111000000000000, %0000000000000000, %0000000000001110,
            %1110000111111100, %0011110001111000, %0000000111100111,
            %1100000111100111, %0011110000111100, %0000001111000011,
            %1100000111100111, %0000000000111100, %0000001111000011,
            %1100000111111100, %0111110000111100, %0011111111000011,
            %1100000111100111, %0011110000111100, %0111001111000011,
            %1100000111100111, %0011110000111100, %0111001111000011,
            %1100000111111100, %0111111001111110, %0011111111100011,
```

```
                %1110000000000000, %0000000000000000, %0000000000000111,
                %0111000000000000, %0000000000000000, %0000000000001110,
                %0001111111111111, %1111111111111111, %1111111111111000);
    { Image-Struktur }
    Img:=Image(0,0,      { keine Verschiebung }
               48,       { Breite }
               12,       { Höhe }
               1,        { nur eine Plane }
               Dat,      { Bilddaten }
               1,0,      { weißes Bild, schwarzer Hintergrund }
               Nil);     { kein weiteres Image }
    DrawImage(Win^.RPort, ^Img, 20, 20);   { Bild ausgeben }
    Delay(4*50);
    Close_Window(Win)
End.
```

2.5 Messages und Signale

Tasks können untereinander Informationen austauschen, indem sie sich gegenseitig "Messages" senden. Das geht im Prinzip so:

- Der empfangende Task muß einen "Messageport" besitzen, was so eine Art softwaremäßiger Briefkasten ist und rein praktisch gesehen wieder mal eine Struktur ist, die irgendwo im Speicher 'rumliegt.
  Natürlich gibt es einen Datentypen "MessagePort" (in der Datei "exec/ports.h"), aber der kann Ihnen ziemlich egal sein: Wenn Sie ein Fenster öffnen, wird automatisch ein solcher Messageport eingerichtet, wo Sie die das Fenster betreffenden Nachrichten "abholen" können (die Nachrichten enthalten Informationen über Betätigung von Gadgets usw.). Dieser Messageport heißt "UserPort", und das gleichnamige Feld in der "Window"-Struktur ist ein Pointer darauf.
- Der sendende Task schickt nun eine Message an den Messageport: der Port enthält eine Liste der empfangenen Messages, in die die Message (natürlich auch irgendwie eine Struktur) eingefügt wird.
- Nun muß der empfangende Task sich diese Nachricht nur noch abholen. Dazu gibt es eine KICK-PASCAL-Function "Get_Msg(port)" von Typ "Ptr", die eine Nachricht aus dem Port abholt und einen Zeiger darauf zurückgibt (bzw. Nil, wenn gerade keine Nachricht da war).
- Als letztes gibt der empfangende Task mit der Prozedur "Reply_Msg" bekannt, daß er die Message gelesen hat.

Sie können also mit

```
Nachricht:=Get_Msg(Win^.UserPort)
```

eine Nachricht im Userport des Fensters abholen. Wenn der Zeiger "Nachricht" nachher nicht "Nil" ist, zeigt er auf eine Struktur des Typs "IntuiMessage", die wie folgt aufgebaut ist (und in der Datei "intuition/intuition.h" definiert wird):

```
IntuiMessage=Record
  ExecMessage:Message;
  Class:Long;
  Code:Word;
  Qualifier:Word;
  IAddress:Ptr;
  MouseX,MouseY:integer;
  Seconds,Micros:Long;
  IDCMPWindow:p_Window;
  SpecialLink:p_IntuiMessage
End;
```

Das erste Feld enthält die eigentliche Message-Struktur, denn solche Messages werden auch zu vielen anderen Zwecken verwendet und vom Betriebssystemteil "Exec" verwaltet. Da aber "Get_Msg" Ihnen eine vollständig initialisierte Intuition-Message zurücksendet, reicht es, wenn wir hier die Felder betrachten, die wichtige Informationen enthalten!, zum Beispiel "Class". "Class" enthält die IDCMP-Flags, die das Ereignis klassifizieren.

Beispiel: Wenn Sie beim Öffnen des Fensters als IDCMP-Parameter die Zahl $200 für "_CLOSEWINDOW" angegeben haben und das Close-Gadget angeklickt wird, erhält Ihr Programm am UserPort des Fensters eine Message, in der das "_CLOSE-WINDOW"-Flag gesetzt ist.

Dazu ein Beispielprogramm:

```
Program IntMsgDemo; { erste Version }

{$incl 'intuition.lib' }

Var
  Win: ^Window;
  Msg: ^IntuiMessage;

Begin
   Win:=Open_Window(20,10, 200,80, 1, _CLOSEWINDOW,
            ACTIVATE or WINDOWCLOSE or WINDOWDEPTH or WINDOWDRAG,
            'Schließ mich!', Nil, 100, 20, 640, 256);
   Repeat

     Repeat        { auf Message warten }
       Msg:= Get_Msg(Win^.UserPort)
     Until Msg<> Nil;

   Until Msg^.Class = _CLOSEWINDOW;
   Close_Window(Win)
End.
```

Hier geben wir erstmals die IDCMP- und Windowflags mit ihren Namen an. Das ist möglich, weil sie in "intuition/intuition.h" (wo sonst?) unter diesen Namen im "CONST"-Teil definiert werden.

Übrigens: es war die Rede davon, daß Messages der Kommunikation zwischen Tasks dienen. In diesem Fall ist der empfangende Task natürlich das PASCAL-Programm. Aber wer sendet? Im AMIGA laufen ständig jede Menge Prozesse, die

für die Verwaltung der Peripherie sorgen und "Devices" (also "Geräte") heißen. Ein solches Gerätetreiberprogramm ist das "input.device", das alle Eingaben über Maus, Joystick und Tastatur verwaltet, allerdings zum Teil Aufgaben an andere Devices weitergibt, z. B. Tastendrücke an das "console.device" oder Joystickbewegungen an das "gameport.device". Dieses Input.-Device ist es, das feststellt, ob und in welchem Fenster eine Maustaste gedrückt wurde und dann dem zugehörigen Task eine Message zukommen läßt.

Vielleicht ist es Ihnen schon aufgefallen, daß wir hier wieder die Todsünde der AMIGA-Programmierung begangen haben: Eine Schleife, die auf eine Message wartet. Es geht auch anders: nämlich mit der KICK-PASCAL-Funktion "Wait_Port(Port)". Sie wartet am angegebenen Messageport auf das Eintreffen einer Nachricht, ohne dabei in nennenswertem Maße Rechenzeit zu verbrauchen, und gibt einen Zeiger auf die Nachricht zurück, ohne sie aber wirklich abzuholen (so daß man nachher noch "Get_Msg" anwenden muß). Wenn am Port bereits eine Nachricht vorliegt, wird natürlich nicht gewartet.

Wir können also in obigem Programm die innere Repeat-Schleife ersetzen durch:

```
Msg:=Wait_Port(Win^.UserPort);
Msg:=Get_Msg(Win^.UserPort);
```

Ferner werden wir noch ein zweites Ereignis abfragen: "MOUSEBUTTONS". Damit hat man die Möglichkeit, festzustellen, ob eine Maustaste im Fenster gedrückt wurde. Die Felder "MouseX" und "MouseY" der in diesem Fall abgeschickten IntuiMessage-Struktur geben die Koordinaten des Mausklicks an, "Code" enthält Informationen, um welche Taste (links oder rechts) es sich handelt und ob sie gedrückt oder losgelassen wurden. Das Programm setzt mit der "PrintIText"-Prozedur einen Stern an alle Stellen, auf die geklickt wird.

Und noch eine weitere Neuerung enthält das Programm: Die KICK-PASCAL-Procedure "Reply_Msg(Mess)". Sie "beantwortet" eine Nachricht, d. h. sie teilt dem sendenden Task mit, daß die Nachricht empfangen und ausgewertet wurde. Auch wenn es nicht unbedingt notwendig ist, sollte man nach erfolgreicher Benutzung "Get_Msg" immer "Reply_Msg" aufrufen.

```
Program IntMsgDemo2; { zweite Version }

{$incl 'intuition.lib' }

Var
  Win: ^Window;
  Msg: ^IntuiMessage;
  Ende: Boolean;

Procedure Star(x, y: integer);
   Var IText: IntuiText;
   Begin
     IText:=IntuiText(3, 0, 0, -4, -4, Nil, '*', Nil);
     PrintIText(Win^.RPort, ^Itext, x, y)
   End;
```

```
Begin
   Win:=Open_Window(20,10, 320,80, 1, _CLOSEWINDOW or MOUSEBUTTONS,
              ACTIVATE or WINDOWCLOSE or WINDOWDEPTH or WINDOWDRAG,
              'Schließ mich! Version 2.0', Nil, 100, 20, 640, 256);
   Ende:= false;
   OpenLib(IntBase, 'intuition.library', 0);   { ist nötig, weil die
                     Intuition-Funktion "PrintIText" benutzt wird. }
   Repeat
    { auf Nachricht warten: }
     Msg:= Wait_Port(Win^.UserPort);
     Msg:= Get_Msg(Win^.UserPort);

    Case Msg^.Class Of
      _CLOSEWINDOW: Ende:=true;
      MOUSEBUTTONS: Star(Msg^.MouseX, Msg^.MouseY)
      Otherwise        { unbekannte Message empfangen }
    End;

    Reply_Msg(Msg);    { Nachricht als ausgewertet kennzeichnen }

  Until Ende;
  Close_Window(Win);
  CloseLib(IntBase);  { eigentlich überflüssig }
End.
```

Das ist doch schon ganz ordentlich. Die Prozedur "Star(x,y)" malt (oder besser: schreibt) einen roten Stern in das Fenster.

Es ist übrigens enorm wichtig, daß der zugehörige Messageport noch existiert, wenn "Reply_Msg" aufgerufen wird. Das bedeutet, daß Sie eine Message, die Sie am UserPort eines Windows empfangen haben, auf keinen Fall nach dem Schließen des Windows beantworten dürfen, denn "Close_Window" entfernt auch den UserPort des zu schließenden Fensters.

Die nächste Verbesserung unseres Programms unterscheidet auch noch zwischen linker und rechter Maustaste. Dazu muß bei den Windowflags das Bit 16 (namens RMBTRAP) gesetzt werden, damit auch die rechte Maustaste eine Message erzeugt (dann können Sie aber keine Pull-Down-Menüs verwenden!). Außerdem wird im folgenden Programm eine weitere Funktion zur Message-Handhabung verwerdet: "Wait(Signals)".

Stellen Sie einmal vor, Sie haben mehrere Fenster geöffnet und wollen in beiden Messages entgegennehmen. Mit "Wait_Port" kann man aber nur an einem Port warten, während alle anderen Ereignisse ignoriert werden. Hier hilft "Wait": Neben der Kommunikation über Messages gibt es noch das eng damit verbundene Signal-System. Ein Signal ist einfach nur ein Bit, daß einem anderen Task übergeben wird. Immer, wenn ein Task eine Nachricht erhält, wird auch ein Signal übermittelt. Die Funktion "Wait" bekommt nun als Parameter ein Langwort, das die Signalbits enthält, auf die gewartet werden soll, und gibt als Ergebnis die empfangenen Signale zurück.

Leider haben die Signalbits und der Wert, den "Wait" zurückgibt, nichts direkt mit den IDCMP-Flags zu tun, sondern folgen einem anderen System. Nach "Wait" ist noch nicht bekannt, an welchem Port die Nachricht eingetroffen ist. Man muß dann alle in Frage kommenden Ports mit "Get_Msg" abklappern.

Ein wichtiger Unterschied zwischen "Wait" und "Wait_Port": "Wait" wartet wirklich, auch dann, wenn im Augenblick des Aufrufs bereits an einem Port eine Nachricht vorliegt. Sie müssen also immer erst prüfen, ob eine Message eingetroffen ist, und erst im negativen Fall "Wait" aufrufen.

Beispiel: Angenommen, Sie haben drei Ports, auf die die Pointer "Port1" bis "Port3" zeigen. Eine Message-Handhabung könnte dann so aussehen:

```
Repeat
  Mess1:=Get_Msg(Port1);
  Mess2:=Get_Msg(Port2);
  Mess3:=Get_Msg(Port3);
  If (Mess1=Nil) and (Mess2=Nil) and (Mess3=Nil) Then
    Signale:=Wait(Signalmaske);  { keine Nachricht da, also warten }
Until (Mess1<>Nil) or (Mess2<>Nil) or (Mess3<>Nil);
```

Nach dieser Schleife hat mindestens (!) eine der Variablen "Mess1" bis "Mess3" einen von NIL verschiedenen Wert. Als "Signalmaske" können Sie im Zweifelsfall $ffffffff bzw. -1 angeben. Dann wird auf ein beliebiges Signal gewartet. Eleganter ist es aber, wenn Sie das Feld "mp_SigBit" aus der MessagePort-Struktur benutzen, das die Nummer des zum Messageport gehörenden Signalbits enthält. Man braucht aber nicht die Nummer des Bits, sondern eine Zahl, in der das entsprechende Bit gesetzt ist. Eine solche Zahl erhält man, indem man die Zahl 1 um die Bitnummer nach links schiebt:

```
Signale:= Wait (Long(1) shl Port^.mp_SigBit)
```

Im obigen Programmfragment müßte man, da man es ja mit drei Ports zu tun hat, drei Bits mit OR verknüpfen, etwa durch folgende Zuweisung:

```
Signalmaske:= (Long(1) shl Port1^.mp_SigBit) or (Long(1) shl
              Port2^.mp_SigBit)
```

Also nun die dritte Version unseres Programms:

```
Program IntMsgDemo3; { dritte Version }

{$incl 'intuition.lib' }

Const
  Rahmenbreite=2;
  Balkenhoehe=12;

Var
  Win: ^Window;
  Msg: ^IntuiMessage;
  Ende: Boolean;
  Signale: Long;
```

```
Procedure Star(x, y, farb: integer);
   Var IText: IntuiText;
   Begin
     IText:=IntuiText(farb, 0, 0, -4, -4, Nil, '*', Nil);
     PrintIText(Win^.RPort, ^Itext, x-Rahmenbreite, y-Balkenhoehe)
   End;

Begin
   Win:=Open_Window(20,10, 200,80, 1, _CLOSEWINDOW or MOUSEBUTTONS,
                    ACTIVATE or WINDOWCLOSE or WINDOWDEPTH
                    or WINDOWDRAG or RMBTRAP or GIMMEZEROZERO,
                    'Klick mich!', Nil, 100, 20, 640, 256);
   Ende:= false;
   OpenLib(IntBase, 'intuition.library', 0);   { ist nötig, weil die
                       Intuition-Funktion "PrintIText" benutzt wird. }
   Repeat

    { auf Nachricht warten: }
    Repeat
      Msg:= Get_Msg(Win^.UserPort);
      If Msg=Nil Then
         Signale:=Wait(1 shl Win^.Userport^.mp_SigBit)
    Until Msg<>Nil;

    Case Msg^.Class Of
      _CLOSEWINDOW: Ende:=true;
      MOUSEBUTTONS: If (Msg^.Code and $80)=0 Then
                     Begin
                        If (Msg^.Code and $01)=0 Then
                          Star(Msg^.MouseX, Msg^.MouseY,3) { links }
                        Else
                          Star(Msg^.MouseX, Msg^.MouseY,1) { rechts }
                     End
      Otherwise
         { unbekannte Message empfangen }
    End;

    Reply_Msg(Msg);    { Nachricht als ausgewertet kennzeichnen }

   Until Ende;
   Close_Window(Win);
   CloseLib(IntBase);  { eigentlich überflüssig }
End.
```

Mit diesem Programm können Sie mit der linken Maustaste wie bisher rote und mit der rechten weiße Sterne malen. Ferner wird ein Gimmezerozero-Fenster benutzt, so daß man den Rahmen nicht mehr mit Sternen übermalen kann. Leider liegt jetzt der Nullpunkt des Koordinatensystems in der Fensterfläche, aber die Messages beziehen sich nach wie vor auf die Ecke des Rahmens. Deshalb müssen die Koordinaten um die Rahmenbreite (2 Punkte) bzw. die Balkenhöhe (12 Punkte) korrigiert werden.

Zum Feld "Code" der Intuition-Message: das siebte Bit (Wert: $80) gibt an, ob die Taste niedergedrückt (1) oder losgelassen (0) wurde (denn beides verursacht eine Message), während das unterste Bit die Maustaste angibt: 0=links, 1=rechts.

2.6 Gadgets

Ein Gadget ist, um es primitiv zu sagen, etwas, wo man mit der linken Maustaste draufklickt und was dann eine Wirkung hat. Intuition kennt verschiedene Gadgettypen: da wären die System-Gadgets (wie die Fensterschließ-, Größenänderungs- oder Front/Back-Gadgets), die sog. Boolean-Gadgets, die Textgadgets, in denen ein Text eingegeben werden kann und die Proportionalgadgets, bei denen ein Schieber horizontal und/oder vertikal verschoben werden kann. Wir wollen uns hier auf die Boolean-Gadgets beschränken, die so heißen, weil sie einfach nur aktiviert oder nicht aktiviert sein können. Das Fenster-Schließ-Gadget ist im Prinzip auch so ein Boolean-Gadget.

Wie Sie sich fast schon denken können, muß auch hier ein Record initialisiert werden. Der Typ heißt (natürlich) "Gadget" und wird (natürlich) in der Datei "intuition/intuition.h" definiert.

So sieht die Gadget-Struktur aus:

```
Gadget = Record
   NextGadget: p_Gadget;            { Verbindungszeiger }
   LeftEdge, TopEdge: integer;      { Position }
   Width, Height: integer;          { Größe }
   Flags: word;                     { diverse Gadget-Eigenschaften }
   Activation: word;                { Art der Aktivierung }
   GadgetType: word;                { Gadgettyp ("1" für Boolean) }
   GadgetRender: Ptr;               { Zeiger auf Text oder Border }
   SelectRender: Ptr;               { Zeiger auf Image, das bei
                                      Aktivierung ausgegeben werden soll}
   GadgetText: Ptr;                 { Zeiger auf Textstruktur }
   MutualExclude: Long;             { gegenseitiger Ausschluß von Gad-
                                      gets}
   SpecialInfo: Ptr;                { zeigt auf zusätzliche Daten für
                                      String- oder Proportionalgadgets }
   GadgetID: integer;               { Identifikations-Nummer }
   UserData: Long                   { Inhalt beliebig }
 End;
```

Der Zeiger "NextGadget" wird von Intuition benutzt, um Gadgets zu einer Liste zu verketten. "LeftEdge" und "TopEdge" geben die Koordinaten der linken oberen Ecke des Gadgets an.

"Width" und "Height" geben die Größe der sog. Hit-Box an. Das ist der Bereich, in dem das Gadget auf Mausklicks reagieren soll, während das Gadget selbst wesentlich größer oder kleiner sein kann. Somit geben genaugenommen "LeftEdge" und "TopEdge" nicht die Position des Gadgets, sondern die der Hit-Box an. Hier ist eine Liste der Gadget-Flags:

Bit	Wert	Name	Bedeutung
0	$0001	GADGHBOX	Gadget wird bei Aktivierung eingerahmt.
1	$0002	GADGHIMAGE	Bei Aktivierung soll ein Image ausgegeben werden (auf das "SelectRender" zeigt).
2	$0004	GADGIMAGE	"GadgetRender" zeigt auf ein Image (sonst wird es als Border interpretiert).
3	$0008	GRELBOTTOM	"TopEdge" bezieht sich auf den unteren Fensterrand.
4	$0010	GRELRIGHT	"LeftEdge" bezieht sich auf rechten Rand.
5	$0020	GRELWIDTH	Breite ist von Fensterhöhe abhängig.
6	$0040	GRELHEIGHT	Höhe ist von Fensterhöhe abhängig.
7	$0080	SELECTED	Flag: Gadget angewählt?
8	$0100	GADGDISABLED	Anwahl des Gadgets unmöglich.

Die beiden untersten Bits geben also an, was passieren soll, wenn das Gadget angeklickt wird. Wenn beide Bits gelöscht sind (entspricht der Konstanten GADGHCOMP, die den Wert 0 hat), wird das Gadget bei Aktivierung invertiert; falls beide Bits gesetzt sind (was der Konstanten GADHNONE=3 entspricht), so passiert nichts Sichtbares.

"GADGIMGE" gibt an, ob die Struktur, auf die "gadgetRender" zeigt, ein Image oder ein Border sein soll. Die Flags, deren Namen mit "GREL" ("Gadget relative") anfangen, sind interessant, wenn die Größe des Fensters variabel ist: dann kann man veranlassen, daß die Größe des Gadgets von der Fenstergröße oder die Position vom rechten oder unteren Fensterrand abhängen soll. Das Bit "SELECTED" gibt an, ob das Gadget gerade aktiv ist. Dagegen bewirkt "GADGDISABLE", wenn es gesetzt ist, daß das Gadget nicht anwählbar ist.

Der nächste Eintrag der Gadget-Struktur heißt "Activation" und besteht wieder aus vielen Bits, die angeben, auf welche Weise das Gadget aktiviert werden kann. Hier ist eine Auswahl der Bits:

Bit	Wert	Name	Bedeutung
0	$0001	RELVERIFY	Gadget wird nur aktiviert, wenn die Maustaste auch in der Hit-Box losgelassen wird.
1	$0002	GADGIMMEDIATE	Gadget wird sofort aktiviert, wenn es angeklickt wird.
3	$0008	FOLLOWMOUSE	Solange das Gadget aktiviert ist, wird bei jeder Mausbewegung deren Position gemeldet. Dadurch können Sie z. B. ein verschiebbares Gadget realisieren.
8	$0100	TOGGLESELECT	Zustand des Gadgets wird bei jedem Anklicken zwischen "aktiviert" und "nicht aktiviert" umgeschaltet.

Das Feld "GadgetType" gibt natürlich den Typen des Gadgets an. Wir beschränken uns auf den Typ 1: "BOOLGADGET". "GadgetRender" zeigt, wie bereits erwähnt, in Abhängigkeit vom GADGIMAGE-Bit entweder auf eine Border- oder eine Image-Struktur. Ebenso zeigt "SelectRender" auf ein Border oder Image. Dieses wird ausgegeben, wenn das Gadget aktivert wird. Der Eintrag "GadgetText" zeigt auf eine Textstruktur ("intuiText"), mit der man das Gadget beschriften kann. Mit "MutualExclude" kann man erreichen, daß die Aktivierung mehrerer Gadgets sich ausschließt. Diese Option soll uns hier aber nicht interessieren, ebensowenig der Zeiger "SpecialInfo", der bei Boolean-Gadgets keine Bedeutung hat, und "UserData", der überhaupt keine Bedeutung hat. Auch in "GadgetID" können Sie prinzipiell einsetzen, was Sie wollen.

Viele dieser Optionen werden im nächsten Beispielprogramm demonstriert. Zuvor muß ich Ihnen aber noch sagen, wie man feststellen kann, ob ein Gadget angeklickt wurde. Auch in diesem Fall erhält das Programm auf die bekannte Weise eine Message des Typs "GADGETUP" oder "GADGETDOWN", (je nach dem, ob das Gadget durch Niederdrücken oder Loslassen der Maustaste aktiviert wurde), wobei der Pointer "IAddress" der IntuiMessage-Struktur auf das Gadget zeigt (das Sie dann z. B. anhand der GadgetID identifizieren können).

```
Program GadgetDemo;
{$include 'intuition.lib' }

Type
  BorDatTyp = Array[1..10] of integer;  { Typ für Borderdata }
  ImgDatTyp = Array[1..12] of Long;     { Typ für Imagedata }
Var
  Win                          : ^Window;
  Msg                          : ^IntuiMessage;
  Gad1, Gad2, Gad3, Gad4, Gad5: Gadget;
  Tex1, Tex2, Tex3, Tex4, Tex5: IntuiText;
  Bor1, Bor2,             Bor5: Border;
  BDt1, BDt2,             BDt5: BorDatTyp;
  Img1, Img2, Img3, Img4      : Image;
  IDt1, IDt2, IDt3, IDt4      : ^ImgDatTyp;
  AktGad                      : ^Gadget;
  Ende                        : Boolean;

Procedure Ausgabe(t: Str);
  { gibt den Text "t" im Window aus }
  Var it: IntuiText;
  Begin
   it:=IntuiText(1,0,1,0,0,Nil,t,Nil);
   PrintIText(Win^.RPort, ^it, 5, 12)
  End;

Begin
  OpenLib(IntBase, 'intuition.library', 0);
  Win:=Open_Window(0, 0, 320, 120,1,_CLOSEWINDOW+GADGETUP+GADGETDOWN,
       WINDOWSIZING+WINDOWDRAG+WINDOWDEPTH+WINDOWCLOSE+ACTIVATE,
        'GadgetDemo', Nil, 300, 100, 640, 256);
```

```
Gad1:=Gadget(Nil,                { erste Gadget-Struktur: }
             10,25,              { Position }
             80,12,              { Größe }
             GADGHCOMP,          { Gadget wird invertiert }
             TOGGLESELECT+GADGIMMEDIATE, { Activation Flags }
             BOOLGADGET,         { Typ }
             ^Bor1,              { Zeiger auf Borderstruktur }
             Nil,                { kein Select-Border }
             ^Tex1,              { Zeiger auf Text }
             0, Nil, 1, 0); { Nummer 1 }
Tex1:=IntuiText(1, 0, 0, 16, 3, Nil, 'Toggle', Nil);
Bor1:=Border(0, 0, 3, 0, 0, 5, ^BDt1, Nil);
BDt1:=BorDatTyp(-3,-2, 83,-2, 83,14, -3,14, -3,-2);

Gad2:=Gadget(Nil,                { zweite Gadget-Struktur: }
             10,48,              { Position }
             80,12,              { Größe }
             GADGHBOX,           { Gadget wird umrandet }
             RELVERIFY,          { Activation Flags }
             BOOLGADGET,         { Typ }
             ^Bor2,              { Zeiger auf Borderstruktur }
             Nil,                { kein Select-Border }
             ^Tex2,              { Zeiger auf Text }
             0, Nil, 2, 0); { Nummer 2 }
Tex2:=IntuiText(1, 0, 0, 8, 3, Nil, 'Nummer 2', Nil);
Bor2:=Border(0, 0, 1, 0, 0, 5, ^BDt2, Nil);
BDt2:=BorDatTyp(-3,-3, 83,-3, 83,14, -3,14, -3,-3);

Gad3:=Gadget(Nil,                { dritte Gadget-Struktur: }
             200,25,             { Position }
             32,12,              { Größe }
             GADGHIMAGE+GADGIMAGE,     { Gadget hat Bild }
             RELVERIFY+TOGGLESELECT,   { Activation Flags }
             BOOLGADGET,         { Typ }
             ^Img1,              { Zeiger auf Imagestruktur }
             ^Img2,              { Select-Image }
             ^Tex3,              { Zeiger auf Text }
             0, Nil, 3, 0); { Nummer 3 }
Tex3:=IntuiText(1, 0, 0, -100, 3, Nil, 'Mit Bild:', Nil);
IDt1:=Ptr(Alloc_Mem(SizeOf(ImgDatTyp),2));
IDt1^:=ImgDatTyp(%00000000000000000000000111000000,
                 %00000000000000000000111111111000,
                 %00000000000000000011111111111111,
                 %00000000000000000000001111100000,
                 %00000000000000000000001111100000,
                 %11111110000000000000001111100000,
                 %11111111110000000000011111100000,
                 %00000011111000000001111111000000,
                 %00000001111111111111111100000000,
                 %00000000011111111111100000000000,
                 %00000000000000000000000000000000,
                 %00000000000000000000000000000000);
```

```
IDt2:=Ptr(Alloc_Mem(SizeOf(ImgDatTyp),2));
IDt2^:=ImgDatTyp(%00000000000000000000000000000000,
                 %00000000000000000000000000000000,
                 %00000000011111111111100000000000,
                 %00000001111111111111111000000000,
                 %00000011111000000001111111000000,
                 %11111111110000000000011111100000,
                 %11111110000000000000001111100000,
                 %00000000000000000000001111100000,
                 %00000000000000000011111111111111,
                 %00000000000000000000111111111000,
                 %00000000000000000000001111100000,
                 %00000000000000000000000111000000);
Img1:=Image(0,0,32,12,1,IDt1,1,3,Nil);
Img2:=Image(0,0,32,12,1,IDt2,1,3,Nil);

Gad4:=Gadget(Nil,              { vierte Gadget-Struktur: }
             -40,20,           { Position }
             32,12,            { Größe }
             GADGHIMAGE+GADGIMAGE+GRELRIGHT,{ relativ zu rechtem
                                              Rand }
             RELVERIFY+TOGGLESELECT,  { Activation Flags }
             BOOLGADGET,       { Typ }
             ^Img4,            { Zeiger auf Image }
             ^Img3,            { Select-Image }
             ^Tex4,            { Zeiger auf Text }
             0, Nil, 4, 0); { Nummer 4 }
Tex4:=IntuiText(3, 0, 0, -4, 3, Nil, ' *4* ', Nil);
IDt3:=Ptr(Alloc_Mem(SizeOf(ImgDatTyp),2));
IDt3^:=ImgDatTyp(%11100000000000000000000000000111,
                 %00111100000000000000000000111100,
                 %00000111000000000000000111100000,
                 %00000001111000000000011100000000,
                 %00000000001110000011110000000000,
                 %00000000000011111111000000000000,
                 %00000000000001111110000000000000,
                 %00000000000111000011110000000000,
                 %00000000111100000000011100000000,
                 %00000111100000000000000111100000,
                 %00111100000000000000000000111100,
                 %11100000000000000000000000000111);
Img3:=Image(0,0,32,12,0,IDt3,1,0,Nil);
IDt4:=Ptr(Alloc_Mem(SizeOf(ImgDatTyp),2));
IDt4^:=ImgDatTyp(%00000011111111111111111111000000,
                 %00001111110000000000000001111000,
                 %00111100000000000000000000111100,
                 %01110000000000000000000000001110,
                 %11100000000000000000000000000111,
                 %11100000000000000000000000000111,
                 %11100000000000000000000000000111,
                 %11100000000000000000000000000111,
                 %01110000000000000000000000001110,
                 %00111100000000000000000000111100,
                 %00001111110000000000000001111110000,
                 %00000011111111111111111110000000);
Img4:=Image(0,0,32,12,0,IDt4,1,0,Nil);
```

```
  Gad5:=Gadget(Nil,              { fünfte Gadget-Struktur: }
               250,80,           { Position }
               -272, -92,        { Größe }
               GADGHCOMP+GRELWIDTH+GRELHEIGHT,
               RIGHTBORDER+BOTTOMBORDER+RELVERIFY,
               BOOLGADGET,       { Typ }
               ^Bor5,            { Zeiger auf Borderstruktur }
               Nil,              { kein Select-Border }
               ^Tex5,            { Zeiger auf Text }
               0, Nil, 5, 0); { Nummer 5 }
  Tex5:=IntuiText(1, 0, 0, 3, 3, Nil, 'Gadget#5', Nil);
  Bor5:=Border(0, 0, 3, 0, 0, 5, ^BDt5, Nil);
  BDt5:=BorDatTyp(-3,-2, 280,-2, 280,90, -3,90, -3,-2);

  AddGadget(Win, ^Gad1, Nil);
  AddGadget(Win, ^Gad2, Nil);
  AddGadget(Win, ^Gad3, Nil);
  AddGadget(Win, ^Gad4, Nil);
  AddGadget(Win, ^Gad5, Nil);

  RefreshGadgets(Win^.FirstGadget, Win, Nil);

  Ende:=False;
  Repeat
     Msg:=Wait_Port(Win^.UserPort);
     Msg:=Get_Msg(Win^.UserPort);
     Case Msg^.Class Of
       _CLOSEWINDOW: Ende:=True;
       GADGETUP,GADGETDOWN:
               Begin
                 AktGad:=Msg^.IAddress;
                 Case AktGad^.GadgetID Of
                   1: If (Gad1.Flags and SELECTED)<>0 Then
                        Ausgabe('Gadget 1 an.       ')
                      Else
                        Ausgabe('Gadget 1 aus.      ');
                   2:   Ausgabe('Gadget 2 angewählt.');
                   3: If (Gad3.Flags and SELECTED)<>0 Then
                        Ausgabe('Pfeil nach unten   ')
                      Else
                        Ausgabe('Pfeil nach oben    ');
                   4: If (Gad4.Flags and SELECTED)=0 Then
                        Ausgabe('Eingekreist.       ')
                      Else
                        Ausgabe('Durchgestrichen.   ');
                   5: Ausgabe('Unten rechts.       ');
                   Otherwise;
                 End  { inneres CASE }
               End;
       Otherwise;
     End; { of CASE }
     Reply_Msg(Msg);
  Until Ende;

  Close_Window(Win);
  CloseLib(IntBase);
  Free_Mem(Long(IDt1),SizeOf(ImgDatTyp));
  Free_Mem(Long(IDt2),SizeOf(ImgDatTyp));
  Free_Mem(Long(IDt3),SizeOf(ImgDatTyp));
End.
```

Das Programm sieht vielleicht kompliziert aus, aber es besteht zu einem großen Teil aus Initialisierungen von IntuiText-, Border- und Image- Strukturen, also Dinge, die wir bereits kennen. Davon einmal abgesehen, werden nicht weniger als fünf Gadgets im Window dargestellt.

Das erste ist ein recht gewöhnliches Toggle-Gadget, das mit einem Rahmen umgeben ist. Sie können mit der Maus die Invertierung des Gadgets ein- und ausschalten. "Gad2" ist sehr ähnlich, aber hier gibt es keine Umschaltfunktion, und das Gadget wird nicht invertiert, sondern umrandet. Beachten Sie auch, daß Gadget 1 seinen Zustand sofort ändert ("GADGIMMEDIATE"), während das Programm auf Gadget 2 erst reagiert, wenn die Maustaste losgelassen wird ("RELVERIFY"). Die beiden nächsten Gagdets haben statt Borders jeweils ein Bild, und zwar mit Select-Image. Beide sind "TOGGLESELECT", d. h. ihre Zustände und damit ihre Bilder werden umgeschaltet.

Gadget 4 hat die Besonderheit, daß es ("GRELRIGHT") bei einer Größenänderung des Fensters stets konstanten Abstand zum rechten Rand hält. Diesen Abstand gibt das Feld "LeftEdge" an, das einen negativen Wert haben muß, denn das Gadget soll in diesem Fall ja links vom Koordinatennullpunkt (der durch das "GRELRIGHT"-Bit am rechten Rand liegt) erscheinen.

Beim Gadget 5 liegt die linke obere Ecke zwar an einer festen Stelle, dafür ist aber die Größe in Abhängigkeit von der Fenstergröße variabel. Die Breite des Gadgets berechnet sich jeweils aus der Summe des Eintrags "Width" und der Fensterbreite, so daß sich am Anfang eine Breite von -272+320 = 48 ergibt - die Gadgethöhe ergibt sich entsprechend aus "Height" und der Fensterhöhe. Dadurch füllt das Gadget stets den durch das (nur teilweise sichtbaren) Border markierten Bereich aus - bis auf zwei gleichbleibende Abstände zu den Fensterrändern.

Nach den Initialisierungen werden die Gadgets mit der Intuition-Funktion "AddGadget(Window, Gadget, Pos)" dem Fenster zugeordnet. Der erste Parameter dieser Procedure ist stets die Windowhandle, der zweite ein Zeiger auf das Gadget. Der dritte Parameter gibt an, an welcher Position das Gadget in die Gadgetliste des Windows eingefügt werden soll. Wenn man hier "Nil" angibt, wird es an erster Stelle eingehängt.

Nun gehören die Gadgets zwar zum Fenster, sind aber noch nicht sichtbar. Vielmehr muß man Intuitiion erst mit der Funktion "RefreshGadgets(GadgetPtr, WindowPtr, Req)" dazu überreden, die Gadgets auszugeben. Diese Funktion gibt das Gadget, auf das der Zeiger "GadgetPtr" zeigt, und alle in der Gadgetliste nachfolgenden Gadgets des Windows aus (der dritte Parameter hat nur bei Requestern eine Bedeutung). Die "Gadgetliste" eines Fensters ist eine lineare Liste, die alle Gadgetstrukturen des Windows enthält. Um den Anfang dieser Liste zu erhalten (und somit alle Gadgets "refreshen" zu lassen), nehmen wir hier den Listenanfangszeiger aus der Windowstruktur.

Die Messagehandhabung enthält nichts Spektakuläres mehr: wenn die empfangene Nachricht aus der Klasse "GADGETUP" oder "GADGETDOWN" ist, wird aus dem

Feld "IAddress" ein Zeiger auf das Gadget entnommen. Anhand des Feldes "GadgetID" wird dann die Nummer des Gadgets festgestellt.

Achtung: So lange die Gadgets benutzt werden, müssen die Gadget- Strukturen auch vorhanden sein! Sie können z. B. nicht die Gagdets als LOKALE Variable einer Prozedur deklarieren, sie dort initialisieren und dem Fenster zuweisen und dann ins Hauptprogramm zurückkehren - denn dann verschwinden die lokalen Variablen und damit die Gadgetstrukturen. Die Folge wäre wahrscheinlich ein Guru.

SYSTEMPROGRAMMIERUNG

KAPITEL III

TEXTEINGABE

III. Texteingabe

3.1 Tastencodes

Die Tastatur des AMIGA ist im Grunde genommen ein selbstständiger Computer, bestehend aus einem einzigen Chip, der den Prozessor (6502-kompatibel!), etwas RAM und ROM enthält. Dieser Tastaturprozessor fragt die Tasten ab und sendet über eine serielle Schnittstelle die Tastencodes an den AMIGA, wo die Daten von einem IC entgegengenommen werden.

Sie können diese hardwaremäßig empfangenen Zeichen direkt abfragen: die Codes stehen an der Adresse $bfec01. Dabei stellen die 7 oberen Bits die Tastennummer dar, während das unterste Bit angibt, ob die Taste niedergedrückt oder losgelassen wurde. Allerdings haben die Tastennummern absolut nichts mit den ASCII-Codes der Tasten zu tun.

Das folgende Programm fragt direkt die Tastatur ab und entschlüsselt in der Prozedur "Decode" die Tastencodes:

```
Program KeyCodes;
Var
 Zeiger: ^Byte;
 LastKey: Byte;

Procedure Decode(i: integer);
   { Bezeichnung der Taste mit dem Code 'i' ausgeben }
   Begin
    If (i >= $50) and (i <= $59) Then
       write('Funktionstaste ', i-$59)
    Else
      If (i >= $00) and (i <= $0d) Then
        write('~1234567890ß'\'.[i+1])
      Else
        If (i >= $10) and (i <= $1b) Then
          write('QWERTZUIOPÜ+'.[i-$0f])
        Else
          If (i >= $20) and (i <= $2b) Then
            write('ASDFGHJKLÖÄ#'.[i-$1f])
          Else
            If (i >= $30) and (i <= $3a) Then
               write('<YXCVBNM,.-'.[i-$2f])
            Else
              Case i of
                $40: write('Space');
                $41: write('Backspace');
                $42: write('Tab');
                $43: write('Enter');
                $44: write('Return');
                $45: write('Escape');
                $46: write('Delete');
                $4b: write('- im Ziffernblock');
                $4c: write('Cursor Hoch');
                $4d: write('Cursor Runter');
                $4e: write('Cursor Rechts');
```

```
            $4f: write('Cursor Links');
            $5e: write('+ im Ziffernblock');
            $5f: write('Help');
            $60: write('Linkes Shift');
            $61: write('Rechtes Shift');
            $62: write('Caps Lock');
            $63: write('Ctrl');
            $64: write('Linkes Alt');
            $65: write('Rechtes Alt');
            $66: write('C=');
            $67: write('AMIGA');
          Otherwise
            write('Unbekannte Taste Nr. ',i);
          End;
    End;

Begin
  Zeiger:= Ptr($bfec01);
  LastKey:= Zeiger^;
  writeln('Beenden mit RETURN');
  writeln;
  Repeat
    While Zeiger^ = LastKey Do ; { auf andere Taste warten }
       LastKey:= Zeiger^;
    Decode((LastKey div 2) xor $7f);
    If odd(LastKey) Then
       writeln(' gedrückt.')
    Else
       writeln(' losgelassen.');
  Until LastKey=2*$3b
End.
```

Vor Aufruf von "Decode" werden die Bits des Tastencodes mit "xor $7f" invertiert. Dadurch kommen die Tastencodes in eine sinnvollere Reihenfolge. Natürlich ist dies keine saubere Programmierung, aber es ist schnell. Man kann es z. B. in Spielprogrammen brauchen. Auch das Laufzeitsystem von KICK-PASCAL fragt auf diese Weise die F10-Taste ab, um nicht mehr Zeit als unbedingt nötig zu verbrauchen.

Es gibt aber eine Möglichkeit, diese Codes abzufragen und dabei sowohl die Warteschleife zu vermeiden als auch zu gewährleisten, daß die Eingabe im richtigen Fenster erfolgt: Das IDCMP-Flag "RAWKEY".

Beispiel:

```
Program RawKey;
{$incl 'intuition.lib' }
  Var
    Win: ^Window;
    Msg: ^IntuiMessage;
    Key: Byte;

  Procedure Decode(i: integer);
     ... Genau wie im Programm 'KeyCodes' ...
```

```
Begin
  Win:=Open_Window(0, 0, 100, 50, 1, RAWKEY,
                   ACTIVATE+WINDOWDEPTH+WINDOWDRAG,
                   'Rawkey-Demo', Nil, 100, 50, 100, 50);
  Repeat
    Msg:=Wait_Port(Win^.UserPort);
    Msg:=Get_Msg(Win^.UserPort);
    Key:=Msg^.Code;
    Decode(Key and $7f);      { Nur Bits 0 bis 6 }
    If (Key and $80)=0 Then   { Bit 7 testen }
      writeln(' gedrückt.')
    Else
      writeln(' losgelassen.')
  Until Key=$c4;     { Code von 'Return loslassen' }
  Close_Window(Win)
End.
```

Die Prozedur "Decode" kennen Sie ja bereits. Wenn das Fenster eine Message des Typs "RAWKEY" empfängt, steht der Tastencode im Feld "Code" der IntuiMessage. Es gibt aber zwei Unterschiede zur direkten Hardware-Abfrage:

- Die Tastennummer ist hier bereits invertiert.
- Die Nummer steht in den sieben UNTEREN Bits, während das Bit 7 angibt, ob die Taste gedrückt oder losgelassen wurde.

Es gibt noch eine dritte Möglichkeit zur Tastaturabfrage: Statt "RAWKEY" kann man auch "VANILLAKEY" verwenden. Dann enthält "Code" den ASCII-Code der gedrückten Taste. Nachteil: Hier werden nur die Tasten gemeldet, die einem einzelnen Zeichen entsprechen, während alle Tasten, die Escape-Sequenzen bewirken würden (z. B. die Cursortasten) völlig ignoriert werden.

Auch hierzu ein Beispielprogramm:

```
Program VanillaKey;
{$path 'pascal:include/'; incl 'intuition.lib' }
Var
 Win: ^Window;
 Msg: ^IntuiMessage;
 Key: integer;

Begin
 Win:=Open_Window(0, 0, 150, 50, 1, VANILLAKEY,
                  ACTIVATE+WINDOWDEPTH+WINDOWDRAG,
                  'Vanillakey-Demo', Nil, 100, 50, 100, 50);
 Repeat
   Msg:=Wait_Port(Win^.UserPort);
   Msg:=Get_Msg(Win^.UserPort);
   Key:=Msg^.Code;
   writeln(Key:3, chr(Key):2);
 Until Key=13;       { Ascii-Code von Return }
 Close_Window(Win)
End.
```

3.2 Das Console-Device

Devices sind Tasks, die Peripheriegeräte ansteuern. Programme können über Messages mit ihnen kommunizieren und so Ein- und Ausgabeaufträge zu ihnen schicken.

Das "console.device" dient der Ein- und Ausgabe in Windows. Prinzipiell ist die Programmierung von Devices etwas kompliziert. KICK-PASCAL stellt aber einige Prozeduren und Funktionen zur Verfügung, mit denen Sie dieses Device bequem programmieren können.

Das Verfahren ist folgendes:

- Zuerst muß mit Open_Window ein Fenster geöffnet werden.
- Dann kann man die Funktion "Con:=OpenConsole(Windowhandle)" aufrufen. Als Parameter ist ihr die Handle des Windows zu übergeben, das vom Device verwaltet werden soll. Es wird ein Wert des Typs "Ptr" zurückgegeben, der eine Art "Devicehandle" ist.
- Nun können Sie die Function "Ch:=ReadCon(Conhandle)" und die Procedure "WriteCon(Conhandle,String)" für Ein- und Ausgaben benutzen.
- Vor dem Schließen des Fensters ist das Console-Device mit der KICK-PASCAL-Prozedur "CloseConsole(Conhandle)" zu schließen.

Mit "WriteCon" können Strings (auch "Str") und Chars, aber keine Zahlen o. ä. ausgegeben werden. Hier kann man auch Escape-Sequenzen anwenden.

Einige Bemerkungen zu "ReadCon": Diese Funktion gibt als Ergebnis einen von der Tastatur empfangegen ASCII-Code zurück. Wurde seit dem letzten "ReadCon" kein Zeichen empfangen, wird "Chr(0)" zurückgegeben. Die Eingabe entspricht im wesentlichen der aus "RAW:"-Files, aber mit dem Unterschied, daß "ReadCon" nicht wartet. Zum Warten auf einen Tastendruck kann (zur Vermeidung einer Schleife) die Funktion "Wait(-1)" verwendet werden.

Ein Beispielprogramm:

```
Program ConsoleDemo;

Const
   Esc = chr($1b);

Var
   Win, Con: Ptr;
   ch: Char;
   Sig: Long;

Begin
   Win:= Open_Window(0, 0, 640, 160, 1, 0, $1006, 'Console-Demo',
                     Nil, 640, 160, 640, 160);
   Con:= OpenConsole(Win);     { Device öffnen }
```

```
  WriteCon(Con,''\n\e'33mDies ist eine Demonstration des Console-
                                                Devices.'\n\n\e'31m');
  WriteCon(Con,'Bitte drücken Sie die Escape-Taste!  -> ');


  Repeat
     Sig:=Wait(-1);
     ch:=ReadCon(Con);
     If ch<>Esc then
       Begin
         WriteCon(Con, ch);    { eingegebenes Zeichen ausgeben }
         WriteCon(Con, ''\n\n\e'3mNein, nicht diese Taste! -> ')
       End;
  Until ch=Esc;

  CloseConsole(Con);
  Close_Window(Win)
End.
```

Ein Hinweis für AMIGA-Kenner oder alle, die schon im Kapitel VI (Devices) gespickt haben: Eine "Devicehandle" gibt es in Wirklichkeit nicht. Der Wert, den "OpenDevice" zurückgibt, ist ein Zeiger auf eine KICK-PASCAL-interne Struktur, die wie folgt aufgebaut ist:

```
ConsoleHandle = Record
  Link        : ^Ptr;          { nur für PASCAL      }
  Size        : Long;          { KICKPASCAL-interne  }
  Special1, Special2: Ptr;     { Zwecke!    }

  ReadIO      : IOStdReq;      { Standard-IO-Requests zum Lesen... }
  WriteIO     : IOStdReq;      { ...und Schreiben. }

  ReadPort    : MsgPort;       { Reply-Ports zum Lesen... }
  WritePort   : MsgPort;       { ...und Schreiben. }

  Buffer, Pos, Len: Word;      { Zeilenpuffer für READ-Zugriff...}
  Line        : STRING[80]     { nach SetStdIO. }
End;
```

Diese Struktur steht in der Datei "Consolehandle.h". Sie können sie für die verschiedensten Zwecke nutzen, z. B. steht im Feld "ReadPort.mp_SigBit" die Nummer des Signalbits, auf das Sie beim Lesen warten müssen, z. B. so:

```
sig:= Wait(1 shl Con^.ReadPort.mp_SigBit);
```

Dabei muß natürlich die Variable "Con" als "^ConsoleHandle" deklariert sein.

Eine weitere Anwendung: Die Einträge namens "io_Unit" in den beiden IO-Request-Strukturen zeigen auf die "ConUnit"-Struktur des Devices. Diese Struktur ist im Include "devices/conunit.h" enthalten und enthält einige interessante Felder:

```
ConUnit=Record
  cu_MP               : MsgPort;
  cu_Window           : Ptr;          { Windowhandle }
  cu_XCP, cu_YCP      : integer;
  cu_XMax, cu_YMax    : integer;      { Fenstergröße in Zeichen }
```

```
    cu_XRSize, cu_YRSize: integer;
    cu_XROrigin, cu_YROrigin    : integer;
    cu_XRExtant, cu_YRExtant    : integer;
    cu_XMinShrink, cu_YMinShrink: integer;
    cu_XCCP, cu_YCCP    : integer;       { Cursorposition }
    cu_KeyMapStruct     : KeyMap;        { Tastaturbelegung }
    cu_TabStops         : Array[0..79]of Word;  {Tabulatorpositionen}
    cu_Mask             : Short;
    cu_FgPen, cu_BgPen  : Short;         { Vorder- & Hintergrundfarbe }
    cu_AOLPen           : Short;
    cu_DrawMode         : Short;         { Zeichenmodus }
    cu_AreaPtSz         : Short;
    cu_AreaPtrn         : Ptr;
    cu_Minterms         : Array[0..7]of Byte;
    cu_Font             : Ptr;           { Zeichensatz }
    cu_AlgoStyle, cu_TxFlags    : Byte;
    cu_TxHeight, cu_TxWidth, cu_TxBaseline, cu_TxSpacing: Word;
    cu_Modes            : Array[0..2]of Byte;
    cu_RawEvents        : Array[0..2]of Byte
  End;
```

Sie können diese Struktur beispielsweise folgendermaßen auslesen:

```
Var Con: ^ConsoleHandle;
    CU : ^ConUnit;

Begin
  ...  { Fenster + Console öffnen usw. }
  CU:= Ptr (Con^.ReadIO.io_Unit);    { Typwandlung ist hier nötig }
  Writeln('Fensterbreite: ', CU^.cu_XMAX, ' Zeichen.');
  ...
End.
```

2.3 Nützliche Unterprogramme

Es ist natürlich unangenehm, daß man mit dem Console.device nur Strings ausgeben und nur Chars eingeben kann. Deshalb werde ich in diesem Kapitel einige Prozeduren vorstellen, mit denen die Ein- und Ausgaben viel mehr Spaß machen.

Als erstes wäre da eine Prozedur, die eine LongInt-Zahl ausgibt. Sie haben sogar die Möglichkeit, verschiedene Basen (Hex, Dezimal, ...) und die Ausgabebreite anzugeben.

```
Procedure WriteConInt(Con: ptr;    { Devicehandle }
                      i: Long;     { Zahl, die ausgegeben werden soll }
                      b: integer; { Basis }
                      f: integer);{ Mindest-Feldbreite }
 Var
  s: String[40];
  j,k,z,len: integer;
  i2: Long;
```

```
Begin
 j:=40;
 s[40]:=chr(0);     { Nullbyte am Ende }
 i2:=abs(i);
 Repeat
   j:= j-1;
   z:= i2 mod b;    { letzte Ziffer von i2 }
   If z<10 Then
     s[j]:=chr(z+ord('0'))      { Ziffern 0 bis 10 }
   Else
     s[j]:=chr(z-10+ord('A')); { Hexziffern A bis F }
   i2:= i2 Div b;
 Until i2=0;
 If b=16 Then
   Begin
     j:=j-1;
     s[j]:='$'      { Hexzahlen automatisch mit "$" }
   End;
 If b=2 Then
   Begin
     j:=j-1;
     s[j]:='%'      { Binärzahlen mit "%" }
   End;
 If i<0 Then
   Begin
     j:=j-1;
     s[j]:='-'      { Minuszeichen bei neg. Zahlen }
   End;
 len:=40-j;         { Gesamtlänge der Zahl }
 For k:=1 to f-len Do
   WriteCon(Con, ' ');      { Am Anfang mit Spaces auffüllen }
 WriteCon(Con, Str(^s[j])) { String ab j-tem Zeichen ausgeben }
End;
```

Die folgende Prozedur liest einen String ein:

```
Procedure ReadConString(Con:Ptr; Var s: String);
  Const
    Backspace = chr(8);
    Return = chr(13);
  Var
    ch: Char;
    i: integer;
    Sig: Long;
  Begin      i:=1;
    Repeat
      Sig:=Wait(-1);
      ch:=ReadCon(Con);
      If ((ch >= chr(32))and(ch < chr(127))) or (ch>=chr(160)) Then
        Begin
          WriteCon(Con,ch);
          s[i]:=ch;
          i:=i+1
        End;
      If (ch=BackSpace) and (i>1) Then
        Begin
          WriteCon(Con,''\8' '\8'); { Ein Zeichen zurück, mit Space
                                      überschreiben und wieder zurück }
```

```
          i:=i-1
        End;
    Until (ch=Return) or (i>=79);
    s[i]:=chr(0);     { mit Space abschließen }
  End;
```

Nun haben wir also einen String eingelesen. Oft will man aber eine Zahl haben. Die folgende Function wandelt einen String in eine Zahl, wobei auch Hexzahlen (mit $) und Binärzahlen akzeptiert werden.

```
Function Convert(s: String): Long;
 Var
   i:Long;
   j, b, z, sign: integer;
 Begin
   i:= 0;
   b:= 10;  { Basis }
   j:= 1;   { Stringanfang }
   While s[j]=' ' Do
     j:=j+1;           { führende Spaces überlesen }
   If s[j]='-' Then
     Begin   { negatives Vorzeichen }
       sign:= -1;
       j:=j+1
     End
   Else
     Begin
       sign:= 1;
       If s[j]='+' Then j:=j+1   { Pluszeichen überlesen }
     End;
   If s[j] = '$' Then   { Hexzahl }
     Begin
       b:=16; j:=j+1
     End;
   If s[j] = '%' Then   { Binärzahl }
     Begin
       b:=2; j:=j+1
     End;
   Repeat
     If (s[j] >= '0') and (s[j] <= '9') Then
       z:= ord(s[j]) - ord('0')
     Else
     If (s[j] >= 'a') and (s[j] <= 'z') Then
       z:= ord(s[j]) - ord('a') + 10
     Else
     If (s[j] >= 'A') and (s[j] <= 'Z') Then
       z:= ord(s[j]) - ord('A') + 10
     Else
       z:= -1;  { ungültige Ziffer }
     If z >= b Then
       z:= -1;  { zu groß für Basis }
     If z >= 0 Then
       i:= b*i + z;
     j:= j+1
   Until z<0;
   Convert:= sign*i
 End;
```

Das nächste Demoprogramm verwendet diese Prozeduren. Es wird ein Fenster nebst Console.device geöffnet und eine Zahl eingelesen, die dann in den drei wichtigsten Basen ausgegeben wird.

```
Program ConsoleToolDemo;

Var
   Win: Ptr;
   Con: Ptr;
   St: String;
   z: Long;

Procedure WriteConInt(Con: ptr; i: Long; b: integer; f: integer);
  ...

Procedure ReadConString(Con:Ptr; Var s: String80);

Function Convert(s: String80): Long; ...

Begin
   Win:= Open_Window(0,0,640,200,1,0,$1006,'Test',
                                          Nil,640,200,640,200);
   Con:= OpenConsole(Win);
   Repeat
     WriteCon(Con, 'Eingabe: ');
     ReadConString(Con, St);
     If St <> '' Then
       Begin
         z:=Convert(St);
         WriteCon(Con,''\n\n); { eine Leerzeile }
         WriteConInt(Con, z, 10, 12); { dezimal, rechtsbündig }
         WriteConInt(Con, z, 16, 12);
         WriteCon(Con,'  ');
         WriteConInt(Con, z,  2,  1); { binär und linksbündig }
         WriteCon(Con, ''\n\n)
       End
   Until St='';          { bei Leerzeile beenden }
   CloseConsole(Con);
   Close_Window(Win)
End.
```

SYSTEMPROGRAMMIERUNG

KAPITEL IV

GRAFIK

IV. Grafik

Die AMIGA-Bibliothek "graphics.library" ist für alles zuständig, was mit Grafik zu tun hat. Sie wird mit den dazugehörigen Datentypen und ihrer Basisvariablen "GfxBase" im Includefile "graphics.lib" deklariert. Wir wollen uns hier auf die wichtigsten Prozeduren und Funktionen beschränken.

Zuerst muß die Grafik-Bibliothek geöffnet werden:

```
OpenLib(GfxBase, 'graphics.library', 0);
```

Die meisten Grafikbefehle beziehen sich auf den uns bereits vertrauten Rastport des Ausgabefensters:

ClearScreen(Rast: Ptr)

Löscht das Fenster, auf dessen Rastport "Rast" zeigt.

Move(Rast: Ptr; x, y: integer)

Setzt den Grafikcursor an die angegebene Position.

Draw(Rast: Ptr; x, y: integer)

Zieht von der Cursorposition in der aktuellen Vordergrundfarbe eine Linie zur angegebenen Stelle.

SetAPen(Rast: Ptr; col: integer)

Bestimmt die aktuelle Vordergrundfarbe. Sie gilt für die meisten Grafikfunktionen, z. B. auch für "Draw".

SetBPen(Rast: Ptr; col: integer)

Legt die Hintergrundfarbe fest (z. B. für Textausgaben).

SetDrMd(Rast: Ptr; mode: integer)

Setzt den Ausgabemodus, z. B. 1 für "normal", 3 für invertierende Ausgabe (die Ausgabe und der Fensterinhalt werden XOR-verknüpft), 4 für inverse Ausgabe (hat nur bei Textausgabe eine Bedeutung).

GfxText(Rast: Ptr; s: Str; count: integer): boolean;

Gibt ab der Cursorposition den Text "s" aus. In diesem Fall muß der String nicht wie gewöhnlich mit einem Nullbyte beendet werden, sondern die auszugebende Länge wird als dritter Parameter übergeben.

Das Flag, das von dieser Funktion zurückgegeben wird, gibt an, ob ein Fehler aufgetreten ist. Eigentlich heißt diese Funktion übrigens schlicht "Text", aber so heißt ja schon ein PASCAL-Datentyp.

RectFill(Rast: Ptr; minX, minY, maxX, maxY: integer);

Füllt ein Rechteck in der Vordergrungfarbe.

WritePixel(Rast: Ptr; x, y: integer);

Setzt einen einzelnen Punkt.

Es gibt noch viel mehr Grafikbefehle, aber dies soll fürs erste genügen.

Ein Demoprogramm für diese Funktionen:

```
Program Grafik;
 {$incl "graphics.lib", "intuition/intuition.h" }
 Var
   Win: ^Window;
   Rp: ^RastPort; { dieser Typ wird in "graphics/rastport.h" definiert}
   err: boolean;
   Msg: Ptr;     i: integer;

Begin
  OpenLib(GfxBase, 'graphics.library', 0);

  Win:= Open_Window(0, 0, 640, 200, 1, _CLOSEWINDOW,
        GIMMEZEROZERO+ACTIVATE+WINDOWCLOSE+WINDOWDRAG+WINDOWDEPTH,
        'Grafikdemo', Nil, 640, 200, 640, 200);
  Rp:= Win^.RPort;

  Move(Rp, 5, 40);
  SetAPen(Rp, 1);     { Ausgabe: weiß ... }
  SetBPen(Rp, 3);     { ...auf rot }
  SetDrMd(Rp, 1);     { damit Hintergrundfarbe auch ausgegeben wird }
  err:= GfxText(Rp, ' Dies ist eine Grafik-Demo. ######', 28);

  SetAPen(Rp, 3);     { rote Linien }
  For i:=0 to 20 Do
    Begin
      Move(Rp, 220+20*i, 5);
      Draw(Rp, 620, 5+9*i)
    End;

  For i:=1 to 10 do
    Begin
      SetAPen(Rp, 1 + i mod 3);     { wechselnde Farben }
      RectFill(Rp, 5*i, 75+2*i, 320-25*i, 180-8*i)
    End;

  Msg:= Wait_Port(Win^.UserPort);  { auf's Schließen warten }
  Msg:= Get_Msg(Win^.Userport);
  Reply_Msg(Msg);
  Close_Window(Win);
End.
```

Noch eine Funktion soll hier erwähnt werden:

SetRGB4

Auf die Dauer wird es natürlich langweilig, immer nur mit den Standard-Bildschirmfarben zu arbeiten. Mit "SetRGB4" können Sie auf Ihren selbstgeöffneten Screens die Farbpalette ändern. Dazu brauchen Sie einen Zeiger auf den "ViewPort" des Screens, der ein Feld (also kein Pointer!) der Screen-Struktur ist. Wenn "Scr" die Screenhandle ist, sieht das so aus:

```
SetRGB4(^Scr^.ViewPort, Nummer, R, G, B)
```

Die Bedeutung der anderen Parameter: "Nummer" ist die Nummer der zu ändernden Farbe. Bei einem Screen mit drei Bitplanes wären hier z. B. Werte von 0 bis 7 sinnvoll.

"R", "G" und "B" geben die roten, blauen und grünen Farbanteile der gewünschten Farbe an. Jeder Anteil ist eine Zahl von 0 bis 15.

Beispiel:

```
... SetRGB4(Vp, 0, 15, 15, 15);  { Hintergrund: weiß    }
    SetRGB4(Vp, 1,  0,  0,  6);  { Farbe 1: Dunkelblau }
    SetREB4(Vp, 2, 15, 11, 11);  { Farbe 2: Rosa        }
    SetRGB4(Vp, 3,  0,  0,  0);  { Farbe 3: Schwarz     }
```

SYSTEMPROGRAMMIERUNG

KAPITEL V

DIE DOS-LIBRARY

V. Die DOS-Library

5.1 Dateibehandlung mit dem DOS

Ein sehr wichtiger Bestandteil des AMIGA-Betriebsystems ist das DOS. Wie Sie sich schon denken können, ist auch dies eine Library. Ihre Basisadresse heißt "DosBase" und ist in KICK-PASCAL eine Standard-Variable, die beim Programmstart automatisch initialisiert wird. Sie müssen die Bibliothek also ausnahmsweise nicht mit "OpenLib" öffnen.

Die Funktionen der Bibliothek werden in der Includedatei "libraries/dos.h" deklariert. Daneben gibt es noch die Datei "libraries/dosextens.h", welche einige zusätzliche Datentypen enthält. Wir werden sie hier aber nicht brauchen.

Viele Funktionen des DOS sind eigentlich überflüssig, da sie nur die Datei-Befehle von PASCAL ersetzen. Die DOS-Funktionen sind aber schneller und flexibler als die PASCAL-Befehle, so daß es lohnenswert ist, sich das DOS einmal näher anzusehen.

Dateien werden mit der Funktion "Handle:=Open(Name, Modus)" geöffnet. "Name" ist ein String und gibt (natürlich) den Dateinamen an. Als Modus kann angegeben werden:

MODE_READWRITE	= 1004	für Ein- und Ausgaben,
MODE_OLDFILE	= 1005	wenn man nur lesen will,
MODE_NEWFILE	= 1006	falls man nur schreiben will.

Die Funktion gibt die "Filehandle" zurück, bzw. den Wert 0, wenn die Datei nicht geöffnet werden konnte. Diese Handle ist aber nicht, wie bei Intuition, ein Zeiger, sondern ein LongInt. Der Grund dafür ist, daß das AMIGA-DOS nicht (wie der Rest des Betriebssystems) in C, sondern in einer Sprache namens BCPL geschrieben wurde. Diese seltsame Sprache behandelt Pointer anders als C oder PASCAL: aus unerklärlichen Gründen ist der Pointer immer ein VIERTEL der Speicheradresse, auf die er zeigt. Wenn Sie also die Filehandle mit vier multiplizieren und dann in einen Zeiger wandeln, erhalten Sie einen Pointer auf die Datei-Struktur. Der Typ "FileHandle" wird übrigens in dem Includefile "libraries/deoextens.h" definiert.
Statt "Open" kann man in KICK-PASCAL auch "DosOpen" schreiben. Dadurch paßt der Name dann zu denen der folgenden Prozeduren:

"DosClose(Fhandle)" schließt eine Datei. Eigentlich heißt diese Systemfunktion schlicht "Close", aber dann würde der Name ja mit der gleichnamigen KICK-PASCAL-Prozedur kollidieren. Ersatzweise können Sie auch "_Close" schreiben.

Wenn Sie eine Datei geöffnet haben, können Sie mit "DosWrite" jede Art von Daten in sie schreiben.

Syntax:

```
Anzahl:= DosWrite(Fhandle, Adr, Anz)
```

"Fhandle" ist wieder die Filehandle, die "Open" zurückgegeben hat. "Adr" ist ein Pointer auf die zu schreibenden Daten, und "Anz" ist die Datenlänge in Bytes. Die Funktion gibt die Anzahl der tatsächlich geschriebenen Bytes zurück. Ist sie von "Anz" verschieden, ist irgendwo ein Fehler aufgetreten. Für den Namen "DosWrite" gilt übrigens sinngemäß dasselbe wie bei "DosClose".

Gelesen wird entsprechend mit der Funktion

```
Anzahl:= DosRead(Fhandle, Adr, Anz)
```

Während "Fhandle" die wohlbekannte Dateihandle ist, geben der Pointer "Adr" und das LongInt "Anz" den Speicher, in den gelesen werden soll, und dessen Länge an. Wieder wird die Anzahl der tatsächlich gelesenen Bytes zurückgegeben. Wenn sie nicht mit der gewünschten übereinstimmt, kann es auch daran liegen, daß das Dateiende erreicht wurde.

Ein Beispielprogramm:

```
Program DosDemo;
{$incl "libraries/dos.h" }
Var
  InFile, OutFile: Long;
  St: String[1000];
  Len, Len2: Long;
Begin
  InFile:= Open('S:Startup-sequence', MODE_OLDFILE);
  If Infile = 0 Then
    Error('Startup-sequence nicht gefunden!');
  OutFile:= DosOpen('ram:copy_of_startup-sequence', MODE_NEWFILE);
  If Outfile = 0 Then
    Begin
      DosClose(InFile); Error('Ausgabedatei nicht geöffnet.')
    End;
  Len:= DosRead(InFile, ^St, 1000);
  If Len = 1000 Then
    writeln('Datei ist länger als 1000 Bytes')
  Else
    If Len = 0 Then
      writeln('Lesefehler oder leere Datei.')
    Else
      Begin
        Len2:= DosWrite(OutFile, ^St, Len);
        If Len2 <> Len Then writeln('Fehler bei Ausgabe!')
      End;
  DosClose(InFile);
  DosClose(OutFile)
End.
```

Dieses Programm liest erst die Startup-Sequence in einem 1000 Bytes langen Puffer und schreibt sie dann in die Ram-Disk.

Weitere DOS-Funktionen:

pos:= DosSeek (Fhandle, Dist, Mode)

Springt in der Datei "Fhandle" an eine Position. Folgende Modi sind möglich:

OFFSET_BEGINNING	= -1	"Dist" ist der Abstand von Dateianfang
OFFSET_CURRENT	= 0	Von der aktuellen Position ist der Lesezeiger um "Dist" Bytes zu verschieben.
OFFSET_END	= 1	Der Offset "Dist" bezieht sich auf das Ende der Datei.

In jedem Fall gibt die Funktion die absolute Position zurück.

fhandle:= DosInput

Diese Funktion liefert die Filehandle der Standard-Eingabedatei. Wenn das Programm vom CLI gestartet wurde, ist dies normalerweise das CLI-Fenster.

fhandle:= DosOutput

Liefert die Standard-Ausgbedatei.

err:= IoErr

Wenn eine DOS-Funktion einen Fehler gemeldet hat, kann man mit dieser Funktion die Fehlernummer abfragen.

err:= DeleteFile(Name)

Löscht eine Datei.

err:= Rename(AltName, NeuName)

Benennt eine Datei um.

err:= Execute(Befehl, Infile, Outfile)

Führt einen CLI-Befehl aus, der als String übergeben wird. Es müssen auch die Ein- und Ausgabedateien angeben werden. Eine "0" entspricht hier jeweils der Standard-Ein- oder -Ausgabe.

5.2 Über Locks und andere verlockende Funktionen

Das Multitasking-Betriebssystem bringt einige Probleme mit sich, von denen MSDOS-Programmierer kaum zu träumen wagen würden. Eines davon ist, wie man verfahren kann, wenn zwei Tasks gleichzeitig auf dieselbe Datei zugreifen wollen.

Das Problem wurde durch sogenannte Locks gelöst. Sie "verriegeln" eine Datei, so daß andere Prozesse nicht mehr oder nur noch eingeschränkt darauf zugreifen können. Es gibt davon gleich zwei Arten:

- Exclusive Lock: Der Task hat die Datei ausschließlich für sich selbst reserviert. Kein anderer Task kann in irgendeiner Weise darauf zugreifen. Übrigens kann der reservierende Task selbst auch kein weiteres Lock darauf einrichten. Eine solche Lock-Struktur ist nötig, wenn auf die Datei schreibend zugegriffen werden soll. Die Konstante "EXCLUSIVE_LOCK", die in der Datei "libraries/dos.h" definiert wird, hat den Wert -1.
- Shared Lock: Der Task will nur lesend auf die Datei zugreifen. Deshalb können noch beliebig viele weitere Shared Locks darauf eingerichtet werden, aber kein Exclusive Lock. Die Konstante "SHARED_LOCK" hat den Wert -2.

Die beiden wichtigsten DOS-Funktionen in diesem Zusammenhang heißen "Lock" und "UnLock":

Lock (Name, Modus): BPTR

Name : String
Modus: EXCLUSIVE_LOCK (-1) oder SHARED_LOCK (-2)

Es wird versucht, auf die durch ihren Namen gegebene Datei bzw. das Directory eine neue Lock-Struktur einzurichten. Bei Erfolg wird ein BCPL-Zeiger (also im Prinzip ein LongInt) darauf zurückgegeben, andernfalls eine Null.
Der Datentyp "BPTR" wird übrigens in der Datei "libraries/dos.h" als "LongInt" deklariert.

UnLock (oldlock)

oldlock : BPTR

Diese Prozedur löscht eine existierende Lock-Struktur. Es ist enorm wichtig, daß nicht mehr benötigte Locks (z. B. am Programmende) stets gelöscht werden, denn sonst bleibt die Datei bzw. das Directory bis zum Reset des ganzen Systems verriegelt.

Ein erstes Beispielprogramm sieht nach, ob der "Dir"-Befehl im "C:"-Directory steht und zum Lesen freigegeben (unverriegelt) ist, indem es versucht, ein Shared Lock darauf einzurichten:

```
Program HalloDir;

{$incl "libraries/dos.h"}

Const Name = 'C:Dir';

Var MyLock: BPTR;

Begin
   MyLock:= Lock (Name, Shared_Lock);
   If MyLock = 0 Then
     Writeln ('Lock konnte nicht eingerichtet werden.')
   Else
     Begin
       Writeln ('DIR-Befehl ist vorhanden und lesbar.');
       UnLock (MyLock)
     End;
 End;
```

Locks sind besonders nützlich, wenn man Informationen über eine Datei oder ein Directory haben möchte. Dazu dient die folgende Funktion:

status:= Examine (lock, InfoBlock)

lock : BCPL-Zeiger (BPTR) auf Lock
InfoBlock: Zeiger auf Pufferspeicher (Typ p_FileInfoBlock)

Der FileInfoBlock wird mit Informationen über die gelockte Datei gefüllt. Es genügt, wenn es sich um ein Shared Lock handelt. Die Variable, auf die "InfoBlock" zeigt, muß aber an einer durch 4 teilbaren Adresse liegen. "Examine" gibt im Fehlerfall eine Null zurück.

Der Datentyp "FileInfoBlock" ist folgendermaßen deklariert:

```
FileInfoBlock = Record
  fib_DiskKey     : Long;         { Diskettennummer }
  fib_DirEntryType: Long;          { Directory: positiv, Datei: negativ }
  fib_FileName    : String[108]; { Dateiname ohne Pfad }
  fib_Protection  : LongInt;      { Protection-Flags ("rwedasph") }
  fib_EntryType   : LongInt;      { Eintragstyp }
  fib_Size        : LongInt;      { Dateilänge in Bytes }
  fib_NumBlocks   : LongInt;      { Anzahl der belegten Blöcke }
  fib_Date        : DateStamp;    { Record mit Datumsangabe }
  fib_Comment     : String[80];   { Kommentar }
  padding         : String[36];   { reserviert }
End;
```

Wir erweitern unser Programm dahingehend, daß wir im Erfolgsfall Informationen über die Datei "Dir" ausgeben. Die Bedingung, daß die Puffervariable für den FileInfoBlock an einer BCPL-tauglichen Adresse liegen muß, erreichen wir, indem wir sie mit "New" dynamisch einrichten:

```
Program DirInfo;

{$incl "libraries/dos.h"}

Const Name = 'C:Dir';

Var MyLock: BPTR;
    FIB   : ^FileInfoBlock;

Begin
   MyLock:= Lock (Name, Shared_Lock);
   If MyLock = 0 Then
     Writeln ('Lock konnte nicht eingerichtet werden.')
   Else
     Begin
       New (FIB);          { Puffer einrichten }
       If Examine (MyLock, FIB) = 0 Then
         Writeln ('Datei konnte nicht untersucht werden')
       Else
         With FIB^ Do    { Informationen ausgeben }
           Begin
             Writeln ('Name:     ', fib_FileName);
             Write   ('Typ:      ');
             If fib_DirEntryType < 0 Then Writeln('Datei')
                                     Else Writeln('Directory');
             Writeln ('Größe     ', fib_Size, ' Bytes');
             Writeln ('          ', fib_NumBlocks, ' Blocks')
           End;
       UnLock (MyLock)           { Lock freigeben }
     End;
 End.
```

Mit Locks kann man auch Directories einlesen. Dazu brauchen wir noch eine weitere Funktion:

status:= ExNext (lock, InfoBlock)

Nachdem ein gelocktes Directory mit "Examine" untersucht wurde, kann man durch wiederholtes Aufrufen von "ExNext" nacheinander über alle darin befindlichen Dateien und Directories Informationen abrufen.

"ExNext" gibt eine 0 zurück, sobald ein Fehler auftrat oder das Ende des Directorys erreicht wurde.

Das folgende Demoprogramm listet das "SYS:"-Verzeichnis auf:

```
Program Directory;

{$incl 'libraries/dos.h' }

Const Name = 'SYS:';

Var MyLock: BPTR;
    FIB   : p_FileInfoBlock;
```

```
Begin
   MyLock:= Lock (Name, Shared_Lock);
   If MyLock = 0 Then
     Writeln ('Lock konnte nicht eingerichtet werden.')
   Else
     Begin
       New (FIB);        { Puffer einrichten }
       If Examine (MyLock, FIB) = 0 Then
         Writeln ('Datei konnte nicht untersucht werden')
       Else
         If FIB^.fib_DirEntryType < 0 Then
           Writeln (Name, ' ist eine Datei')
         Else
           Begin
             Writeln ('Inhalt von ', Name);
             Writeln ('  Size Type Name');
             While ExNext (MyLock, FIB) <> 0 Do
               With FIB^ Do   { Informationen ausgeben }
                 Begin
                   Write (fib_Size:6);
                   If fib_DirEntryType < 0 Then Write (' File ')
                                           Else Write (' Dir  ');
                   Writeln (fib_FileName)
                 End
           End;
       UnLock (MyLock)          { Lock freigeben }
     End;
End.
```

Weitere Lock-Funktionen:

olddir:= CurrentDir (newdir)

newdir: BCPL-Zeiger auf Lockstruktur eines Directorys

Das durch "newdir" spezifizierte Verzeichnis wird (wie durch den CLI-Befehl "CD") zum aktuellen Verzeichnis erklärt. Dabei wird ein BCPL-Zeiger auf die Lockstruktur des bisherigen aktuellen Directorys zurückgegeben.

parent:= ParentDir (lock)

lock: BCPL-Zeiger auf Lock

Es wird Lock auf das Verzeichnis, in dem die duch "lock" gegebene Datei (bzw. Directory) liegt, eingerichtet und zurückgegeben. Durch wiederholtes Aufrufen von "ParentDir" kann man sich so zur Wurzel eines Verzeichnisbaums durchhangeln. "ParentDir" gibt eine Null zurück, wenn der Parameter bereits ein Lock auf das oberste Directory (z. B. "DF0:") war.

Vergessen Sie aber nicht, nach Aufruf von "ParentDir" das als Ergebnis zurückgegebene Shared Lock zu löschen, sonst bleibt das Directory verriegelt.

Die folgende rekursive Prozedur gibt zu einem Lock den vollständigen Pfadnamen aus:

```
Procedure LockName (l: BPTR);
  Var fib    : p_FileInfoBlock;
      parent: BPTR;
  Begin
    parent:= ParentDir (l);
    If parent <> 0 Then                  { gibt es höheres Directory? }
      LockName (parent);                 { dann Rekursion }
    New (fib);
    If Examine (l, fib) <> 0 Then
      Begin
        Write (fib^.fib_FileName);
        If fib^.fib_DirEntryType >= 0 Then { Directory? }
          If parent = 0 Then Write (':')   { Root }
                        Else Write ('/');  { Unterverzeichnis }
      End;
    Dispose(fib);
    If parent <> 0 Then Unlock (parent);
  End;
```

newlock:= CreateDir (Name)

name: String

Ein neues Directory wird samt Lockstruktur erzeugt und ein BCPL-Zeiger darauf zurückgegeben. Es empfiehlt sich, nach "CreateDir" stets "UnLock" aufzurufen, um so das Verzeichnis für die Allgemeinheit zugänglich zu machen.

SYSTEMPROGRAMMIERUNG

KAPITEL VI

DEVICES

VI. Devices

6.1 Allgemeines

Fast alles, was in diesem Buch bisher über die Programmierung des AMIGA-Systems gesagt wurde, befaßte sich in der einen oder anderen Form mit Ein- und Ausgabe von Daten (wozu unter anderem die Behandlung von Dateien, Bildschirmausgaben aller Art oder der Dialog mit dem Benutzer gehören). Dies kann sich für den KICK-PASCAL-User auf vier verschiedenen "Ebenen" vollziehen:

Ebene	Beispiel für Eingabe	Beispiel für Ausgabe
KICK-PASCAL	ReadIn, ReadCon	WriteIn, Page
Libraries	GetMsg, Gadgets	PrintIText, Draw
Devices	???	???
Hardware	$BFEC01 (Tastatur)	$DFF180 (Hintergrundfarbe)

Ihnen wird an obiger Tabelle auffallen, daß wir uns mit der letzten Stufe oberhalb der Hardware-Programmierung, den Devices, noch nicht richtig beschäftigt haben. Zwar kennen Sie schon das Console-Device, aber das wurde bisher nur über KICK-PASCAL-Befehle, also auf der allerobersten Ebene unserer Hierarchie, programmiert. Also dürfen Sie 'mal raten, womit wir uns in diesem Kapitel beschäftigen wollen...

Genau, wir beschreiten den mühsamen Weg der Device-Programmierung.

Was ein Device ist, wurde ja schon im Kapitel über das Consoledevice angedeutet. Was aber braucht man, um ein Device direkt zu programmieren? Einleuchtenderweise geschieht das über die niedrigste Ebene des AMIGA-Betriebssystems, nämlich die Exec-Library, die Ihnen schon bei der Message-Verwaltung begegnet ist. Allerdings "fehlen" im Exec einige geradezu unverzichtbare Funktionen, die den Umgang mit Devices zumindest halbwegs komfortabel machen. Sie sind allgemein als "Exec Support"-Funktionen bekannt und als C-Quelltext in den AMIGA Reference Manuals (siehe Literaturhinweise am Ende dieses Buchs) enthalten. Das bedeutet natürlich nicht, daß Sie jetzt von PASCAL auf "C" unsteigen müssen: KICK-PASCAL 2.0 stellt Ihnen diese Funktionen zur Verfügung; zwar nicht als fertige Befehle, aber als (in PASCAL geschriebenes) Standard-Unit. Es heißt "ExecSupport".

Dann gibt es da noch das Unit "ExecIO". Es benutzt "ExecSupport", lädt in seinem Interface-Teil einige unverzichtbare Includedateien und stellt zwei weitere Prozeduren zur Verfügung - dazu später mehr.

Der Witz an der ganzen Sache mit den Devices ist, daß sie zwar für völlig verschiedene Ein-/Ausgabegeräte zuständig sind, ihre Benutzung aber nach einem absolut identischen Schema erfolgt. Deshalb ist es auch möglich, eine Festplatte oder eine neue RAM-Disk (z. B. "RAD:" oder "VD0:") in das Betriebssystem

einzubinden: Das AMIGADOS kann die jeweiligen Devices genauso aufrufen wie das Trackdisk-Device, das für die Diskettenlaufwerke zuständig ist.

Also verfügen alle Devices über eine bestimmte Menge von Grundoperationen, z. B. Lesen und Schreiben von Daten. Außerdem haben die meisten Devices noch Zusatzfunktionen:

Beim Serial-Device kann man die Baudrate usw. der seriellen Schnittstelle festlegen, das Trackdisk-Device kann Diskettenwechsel feststellen, das Printer-Device kann außer Texten auch Hardcopys von Grafiken drucken und so weiter und so fort. Sie sehen daran schon, daß es oft sinnvoll ist, Devices direkt zu programmieren: alle diese Funktionen stehen Ihnen nicht zur Verfügung, wenn Sie die Geräte mit dem Umweg über das DOS programmieren. Außerdem können Devices (die ja, wie Sie sich erinnern werden, Tasks sind) unabhängig von ihrem "Auftraggeber" arbeiten - der sich während dessen anderen Tätigkeiten widmen kann.

Nach dieser Einleitung sind Sie hoffentlich motiviert genug, um die (zunächst) graue Theorie der Devices ertragen zu können:

6.2 Ports, IO-Requests und andere Formalitäten

Wie Sie bereits (hoffentlich) wissen, erteilen Tasks über Messages Aufträge an Devices, die ebenfalls vollwertige Tasks sind. Also brauchen wir als erstes einen Messageport. Woher nehmen, wenn nicht stehlen? Nun, eine der ExecSupport-Funktionen erzeugt uns einen vollständig initialisierten Port und gibt einen Zeiger darauf zurück:

```
port:= CreatePort (Name, Pri);
```

"Name" ist ein String und gibt den Namen an, den der Port zwecks einfacherer Auffindbarkeit erhalten soll. "Pri" ist ein Byte und gibt die Priorität des Ports an. Sie interessiert uns hier nicht und wird deshalb stets auf 0 gesetzt. Der Rückgabewert der Funktion ist vom Typ "p_MsgPort", also ein Zeiger auf einen Messageport. Konnte CreatePort keinen Port erzeugen, wird ein Laufzeitfehler gemeldet.

Das Gegenstück dazu ist die Prozedur

```
DeletePort (port);
```

Sie löscht einen mit "CreatePort" erzeugten Messageport, auf den "port" zeigt. Man kann diese Prozedur aber ruhig weglassen, denn das Unit "ExecSupport", in dem die beiden Funktionen enthalten sind, sorgt am Programmende mittels eines Exitservers dafür, daß alle erzeugten und noch nicht entfernten Messageports gelöscht werden.

Manchmal richtet man übrigens mehrere Ports für ein Device ein, z. B. dann, wenn man parallel lesen und schreiben will, denn Messages werden ja als Liste verwaltet und folglich werden Aufträge an das Device nacheinander abgefertigt. Ein Beispiel dafür ist die serielle Schnittstelle (serial.device), wo man im Vollduplex-Betrieb auch Daten empfangen will, während man selbst sendet.

Nun haben wir einen Port. Jetzt benötigen wir noch Messages. Bei Devices benutzt man in der Regel nur eine Messagestruktur, die man immer wieder neu initialisiert. Diese Message muß aber eine ganz bestimmte, bei allen Devices gleiche Form haben, welche durch ein Record festgelegt ist, nämlich die IO-Request-Struktur:

```
IOStdReq = Record
    io_Message: Message;
    io_Device : p_Device;
    io_Unit   : p_Unit;
    io_Command: Word;
    io_Flags  : Byte;
    io_Error  : Short;
    io_Actual : Long;
    io_Length : Long;
    io_Data   : Ptr;
    io_Offset : Long
End;
```

Das erste Feld ist die eigentliche Messagestruktur, an die der Inhalt der Nachricht, alle folgenden Record-Felder, angehängt ist.

Die Message braucht uns nicht näher zu interessieren. Erwähnenswert ist nur, daß darin ("Message" ist ein Record-Typ) auch ein Zeiger auf den Messageport eingetragen wird. Wenn wir dann später über spezielle Exec-Funktionen auf das Device zugreifen, müssen wir nur einen Zeiger auf die IO-Request-Struktur als Parameter übergeben - welcher Port gemeint ist, kann Exec dann ja diesem Record entnehmen.

Als nächstes folgen Zeiger auf das Device, das wir anschließend öffnen werden, und eine "Unit"-Struktur. Letzteres hat nichts mit den KICK-PASCAL-Units zu tun, sondern hängt damit zusammen, daß mehrere Tasks gleichzeitig ein Device öffnen können. Es kann sogar ein Task dasselbe Device mehrfach öffnen, z. B. das Trackdiskdevice je einmal für jedes angeschlossene Diskettenlaufwerk. Damit das arme Device, das ja auch nur ein Programm ist, seine verschiedenen "Auftraggeber" auseinanderhalten kann, wird beim Öffnen eines Devices jeweils eine neue sogenannte Unit-Struktur erzeugt. Nach dem ganzen "Gepointere" kommen nun die Felder, die Sie als Device-Anwender benutzen:

Zunächst wäre da "io_Command". In dieses Wort schreibt man die Nummer des auszuführenden Befehls. Auch wenn ein Device so etwas ähnliches wie eine Library ist, werden ihre Funktionen von außen nicht über eine Tabelle von Prozeduren und Funktionen, sondern durch Befehlsnummern aufgerufen. Es gibt bestimmte allgemeine Funktionsnummern, die bei allen Devices gleich sind, und meist noch Nummern für gerätespezifische Operationen. Dazu später mehr.

"io_Flags" enthält gerätespezifische Statusinformaitionen, im Byte "io_Error" gibt das Device evtl. Fehlernummern zurück.

Ein Device dient im wesentlichen der Datenübertragung, ganz egal, ob diese Daten nun ein- oder ausgegeben werden. Diese Daten werden von den folgenden Einträgen der IO-Request-Struktur beschrieben: "io_Data" ist ein Zeiger, der die Anfangsadresse der Daten enthält.

"io_Length" ist die Datenlänge in Bytes. Oft ist es möglich, hier den Wert "-1" zu setzen, die dann einer variablen Datenlänge entspricht.

"io_Actual" wird vom Device beschrieben. Hier steht nach beendeter Operation die tatsächlich übertragene Datenlänge.

"io_Offset" ist gerätespezifisch und enthält immer so etwas wie eine Adressdistanz. Beispielsweise steht hier beim Trackdisk-Device die Nummer des Diskettenblocks.

Oft reichen diese Felder nicht, und so benutzen viele Devices eine IO-Request-Struktur, die um einige gerätespezifische Einträge erweitert ist. Z. B. benötigt das Audio-Device auch Angaben über Frequenz und Lautstärke der abzuspielenden Sounddaten. Deshalb heißt die oben genannte Struktur auch "Standard-IO-Request", im Gegensatz zu den erweiterten "Extended-IO-Request"-Records.

Deshalb enthält das Unit "ExecSupport" gleich zwei Funkionen, die IO-Requests erzeugen:

```
ioreq:= CreateStdIO (ioReplyPort)
```

und

```
ioreq:= CreateExtIO (ioReplyPort, size)
```

Beide Funktionen erzeugen eine solche Struktur und geben Zeiger darauf zurück. Erstere erzeugt eine der oben beschriebenen normalen Strukturen und benötigt dazu nur einen Zeiger auf den Messageport, der wie oben beschrieben in der Message-Struktur eingetragen wird. Letztere benötigt zusätzlich noch die Angabe der Größe (in Bytes) der zu erzeugenden Struktur.

Entsprechend gibt es auch zwei Prozeduren, um solche Records nach Schließen des Device wieder zu löschen:

```
DeleteStdIO (ioStdReq)
```

und

```
DeleteExtIO (ioExtReq).
```

Beide benötigen als Parameter nur einen Zeiger auf die zu löschende IO-Request-Struktur, denn die Größe dieser Struktur, die man bei "CreateExtIO" explizit angeben muß, wird in der Messagestruktur abgespeichert. Auch sonst sind die beiden Prozeduren absolut identisch, es ist also egal, welche man verwendet. Da auch hier am Programmende vollautomatisch gelöscht wird, was noch nicht aufgeräumt ist,braucht man sie überhaupt nicht zu verwenden.

6.3 Wie man endlich ein Device öffnet

Nehmen wir an, daß wir nun einen Port eingerichtet und eine IO-Request-Struktur erzeugt haben. Jetzt können wir last, but not least das Device öffnen, und zwar mit einer Exec-Funktion:

```
fehler:= OpenDevice (name, unit, ioreq, flags)
```

Die Parameter:

name	Str	Device-Name, z. B. 'trackdisk.device'
unit	LongInt	Gerätenummer, z. B. Laufwerksnummer (bei Trackdisk) oder Windowhandle (bei Console)
ioreq	Ptr	Zeiger auf die IO-Request-Struktur, über die wir die gesamte Kommunikation abwickeln wollen.
flags	LongInt	Gerätespezifische Flags, meist "0".

Natürlich gibt es auch dazu ein Gegenstück:

```
CloseDevice (ioReq)
```

Diese Prozedur benötigt als Parameter nur einen Zeiger auf die IO-Request-Struktur, denn "OpenDevice" initialisiert auch die oben beschriebenen Zeiger "io_Device" und "io_Unit", so daß Exec daraus genau ablesen kann, welches Device denn nun geschlossen werden soll.

Mit KICK-PASCAL wird ein Standard-Unit namens "ExecIO" geliefert. Es benutzt zunächst das Unit "ExecSupport" (mit den oben beschriebenen Port- und IO-Request-Funktionen) und lädt im Interfaceteil außerdem noch die Include-Dateien "exec/libraries.h", "exec/devices.h" sowie "exec/io.h" ein, die Sie zur Device-Programmierung brauchen. Ferner stellt es noch zwei weitere Prozeduren zur Verfügung:

```
Open_Device (name, unit, ioreq, flags)
```

und

```
Close_Device (ioReq)
```

Sie sind mehr oder weniger identisch mit den Exec-Funktionen, von denen sie sich nur durch den Unterstrich in der Mitte unterscheiden, bis auf zwei Besonderheiten:

- "Open_Device" steigt mit einer Laufzeit-Fehlermeldung aus, wenn das Device nicht geöffnet werden konnte. Deshalb ist es eine Prozedur und keine Funktion wie "OpenDevice".
- Auch hier wird am Programmende automatisch aufgeräumt, sofern noch nicht geschehen. Auf "Close_Device" kann also durchaus verzichtet werden.

Wir wollen in diesem Handbuch die Prozeduren des Units "ExecIO" benutzen. Aber natürlich können Sie auch genauso gut die entsprechenden Exec-Funktionen verwenden, z. B. wenn Sie einen Fehler bei "OpenDevice" selbst abfangen wollen.

Übrigens: Falls Sie "von Hand" aufräumen wollen, ist stets zuerst das Device zu schließen, erst dann dürfen die IO-Request-Struktur und der Messageport entfernt werden.

6.4 Kommunikation mit Devices - ganz allgemein

Nachden Sie das Device geöffnet haben, können Sie Anweisungen in die IO-Request-Struktur schreiben. In PASCAL könnte das so aussehen:

```
With MyIOReq^ Do
   Begin
     io_Command:= Funktionsnummer;
     io_Data    := ^Puffervariable;
     io_Length  := SizeOf(Puffervariable);
     { evtl. weitere Initialisierungen }
   End;
```

Es gibt einige Standard-Funktionen, über die alle Devices verfügen und deren Nummern in der Datei "exec/io.h" definiert werden:

```
Const
   CMD_RESET  = 1;  { Device-Unit zurücksetzen }
   CMD_READ   = 2;  { Daten lesen }
   CMD_WRITE  = 3;  { Daten schreiben }
   CMD_UPDATE = 4;  { Alle internen Puffer schreiben }
   CMD_CLEAR  = 5;  { Alle internen Puffer löschen }
   CMD_STOP   = 6;  { Jede Tätigkeit des Units unterbrechen }
   CMD_START  = 7;  { Nach "CMD_STOP" weitermachen }
   CMD_FLUSH  = 8;  { Alle Operationen des Units ganz abbrechen }
```

Nun haben wir die IO-Request-Struktur initialisiert und müssen nur noch die Operation auslösen. Dazu gibt es prinzipiell zwei sehr verschiedene Möglichkeiten:

- Synchronisiert: Mit der Exec-Funktion "DoIO" wird ein Auftrag an das Device abgeschickt. Der aufrufende Task ruht, bis das Device fertig ist. Die Funktion gibt eine Fehlernummer (LongInt) zurück, der Rückgabewert "0" entspricht "kein Fehler".
- Unsynchronisiert: Die Exec-Funktion "SendIO" schickt lediglich die Message (IO-Request) an das Device ab. Nun arbeitet das Device gleichzeitig mit dem aufrufenden Programm, man nutzt hier also die Multitasking-Fähigkeit des AMIGA.

Wenn Sie "SendIO" verwenden, können Sie mit der Funktion "CheckIO" feststellen, ob das Device fertig ist, und mit "WaitIO" sauber (ohne Busy Loop) wie bei der synchronisierten Kommunikation auf die Beendigung der E/A-Operation warten.

Alle genannten Funktionen geben eine Fehlernummer zurück und haben nur einen Parameter: einen Zeiger auf die IO-Request-Struktur.

Dann gibt es noch die Exec-Prozedur "AbortIO (ioReq)". Sie bricht bei unsynchronisierter Ein-/Ausgabe die momentane Operation ab.

Eng mit "SendIO" verwandt ist "BeginIO". Diese Funktion ist aber nicht Bestandteil von Exec, sondern die elementare Routine, die jedes Device zur Verfügung stellt. Man kann sie mit Hilfe des Units "ExecSupport" aufrufen. Sie funktioniert im Prinzip genauso wie "SendIO", hat aber weniger "Überbau" seitens des Betriebssystems. Es gibt einige gerätespezifische Gelegenheiten, wo man "BeginIO" statt "SendIO" verwendet.

6.5 Endlich wird's konkret: das Trackdisk-Device

Nach so viel Theorie können wir uns an ein Beispiel wagen ("Daß wir das noch erleben durften!"). Wir wollen den AMIGA-Disketten einmal auf unterster Ebene, nämlich mit dem Trackdisk-Device, zu Leibe rücken und einzelne Blöcke lesen und schreiben (VORSICHT!) und 'mal sehen, was man noch so alles machen kann.

Das Wichtigste ist in diesem Zusammenhang das Unit "ExecIO", welches das Unit "ExecSupport" benutzt, das wiederum "ExecIO" aufruft. Desweiteren brauchen wir noch eine Include-Datei namens "trackdisk.h", das im Include-Verzeichnis unter der Rubrik "devices" steht. Dort stehen auch derartige Includes für die meisten anderen Devices, worin dann jeweils die besonderen Befehlsnummern und evtl. die erweiterte IO-Request-Struktur dieses Geräts stehen.

Also programmieren wir folgendermaßen 'los:

```
Program TrackdiskDemo;

Uses ExecIO;

{$incl 'devices/trackdisk.h' }
```

Wie Sie bereits wissen (sollten), benötigen wir für unser Device zwei Variablen:

```
Var port : ^MsgPort;   { Zeiger auf Message-Port }
    ioreq: ^IOExtTD;   { Zeiger auf erweiterte IO-Request-Struktur }
```

Die Extended-IO-Request-Struktur für das Trackdisk sieht übrigens so aus:

```
IOExtTD = Record
  iotd_Req     : IOStdReq;
  iotd_Count   : Long;
  iotd_SecLabel: Long
End;
```

Sie besteht also aus einer Standard-Struktur und nur zwei zusätzlichen Langworten Mit drei Befehlen öffnen wir nun das Device. Dabei müssen wir sofort festlegen, welches Diskettenlaufwerk wir benutzen wollen; also führen wir dafür einen Parameter ein:

```
Procedure OpenTrackdisk (Laufwerksnummer: integer);
   Begin
     port:= CreatePort ('Disk-Device', 0);
     ioreq:= CreateExtIO (port, SizeOf(IOExtTD));
     Open_Device ('trackdisk.device', Laufwerksnummer, ioreq, 0)
   End;
```

Weil es sauberer ist, schreiben wir auch eine Prozedur, die unser Device wieder schließt:

```
Procedure CloseTrackdisk;
   Begin
     Close_Device (ioreq);
     DeleteExtIO (ioreq);
     DeletePort (port)
   End;
```

Nun können wir schon unsere ersten Anweisungen an das Device schicken. Zunächst wollen wir den Motor ein- und ausschalten. Dies ist eine gerätespezifische Operation, deren Funktionsnummer die Konstante "TD_MOTOR" (steht in der Include-Datei) ist. Als "Länge" wird in der IO-Request-Struktur "1" zum Einschalten und "0" zum Ausschalten eingetragen, das "io_Data"-Feld wird nicht benutzt. Wir wollen hier die Funktion "DoIO" verwenden:

```
Procedure MotorOn;
   Var err: integer;
   Begin
     With ioreq^.iotd_req Do
       Begin
         io_Command:= TD_MOTOR;
         io_Length:= 1;
       End;
     err:= DoIO (ioreq)
   End;

Procedure MotorOff;
   Var err: integer;
   Begin
     With ioreq^.iotd_req Do
       Begin
         io_Command:= TD_MOTOR;
         io_Length:= 0;
       End;
     err:= DoIO (ioreq)
   End;
```

Dann gibt es noch drei Statusfunktionen, mit denen man ermitteln kann, ob eine Diskette im Laufwerk liegt, ob sie schreibgeschützt ist und wie oft sie bisher gewechselt wurde:

```
Function DiskImLaufwerk: Boolean;
  { Stellt fest, ob Disk in dem Laufwerk ist, für das das Device
    geöffnet wurde. }
  Var err: integer;
  Begin
    { Nur ein Befehl ohne Parameter: }
    ioreq^.iotd_req.io_Command:= TD_CHANGESTATE;
    err:= DoIO (ioreq);
    { Rückgabe erfolgt in "io_Actual": }
    DiskImLaufwerk:= ioreq^.iotd_req.io_Actual = 0
  End;

Function Schreibschutz: Boolean;
  { true, wenn Disk schreibgeschützt }
  Var err: integer;
  Begin
    { Nur ein Befehl ohne Parameter: }
    ioreq^.iotd_req.io_Command:= TD_PROTSTATUS;
    err:= DoIO (ioreq);
    { Rückgabe erfolgt in "io_Actual": }
    Schreibschutz:= ioreq^.iotd_req.io_Actual <> 0
  End;

Function Diskwechsel: Long;
  { Gibt Anzahl der bisherigen Diskettenwechsel an. Der Zähler wird
    sowohl beim Einlegen als auch beim Entnehmen einer Disk
    hochgezählt! }
  Var err: integer;
  Begin
    { Nur ein Befehl ohne Parameter: }
    ioreq^.iotd_req.io_Command:= TD_CHANGENUM;
    err:= DoIO (ioreq);
    { Rückgabe erfolgt in "io_Actual": }
    Diskwechsel:= ioreq^.iotd_req.io_Actual
  End;
```

Nun wollen wir aber endlich Daten lesen. Dabei besteht noch das Problem, daß Diskdaten nur in das Chip-RAM gelesen werden können. Wir lösen dies auf die bekannte Weise.

Es gibt aber zwei Lesebefehle: die Standardfunktion "CMD_READ" und die erweiterte Operation "ETD_READ". Letztere wird nur durchgeführt, wenn die Diskette nicht gewechselt wurde, was daran erkannt wird, daß die von "TD_CHANGENUM" zurückgegebene Wechselanzahl größer ist als das Feld "iotd_Count" der Extended-IO-Request-Variablen. Falls ein Wechsel erkannt wurde, wird eine Fehlermeldung zurückgegeben. Bei "CMD_READ", das wir in unserem Beispiel benutzen wollen, wird "iotd_Count" ignoriert.

```
Procedure WriteHex (n: Long; digits:integer);
  { Hexzahl n mit gewünschter Ziffernanzahl ausgeben }
  Begin
    If digits > 1 Then WriteHex (n shr 4, digits-1);
    Write ('0123456789abcdef'.[n and 15 + 1])
  End;
```

```
Procedure LiesBlock (nr: integer);
   { Einen Block lesen, z. B. 0 für Bootblock, 880 für Root..., und
     als kombinierten Hex- und Asciidump ausgeben. }
   Type
     BufferTyp = Array[1..512] Of Byte;  { 1 Block = 512 Bytes }
   Var
     Buffer: ^Buffertyp;
     i, j  : integer;
     err   : integer;
   Begin
     Buffer:= Ptr(Alloc_Mem(SizeOf(BufferTyp),2));  { Chip-RAM }
     With ioreq^.iotd_req Do
       Begin
         io_Command:= CMD_READ;
         io_Data   := Buffer;
         io_Length := 512;
         io_Offset := 512 * nr  { Blockposition in Bytes }
       End;
     err:= DoIO(ioreq);

    { Ergebnis ausgeben: }
     If err <> 0 Then
       Writeln ('Fehler: ', err)
     Else
       Begin
         Writeln ('Inhalt von Block ', nr, ':');
         Writeln;
         For i:=0 to 31 Do     { 32 Zeilen zu 16 Bytes }
           Begin
             WriteHex (16*i,4); Write(': ');
             For j:=1 to 16 Do
               Begin WriteHex (Buffer^[16*i+j], 2); Write (' ') End;
             Write ('  ');
             For j:=1 to 16 Do
               If Buffer^[16*i+j] in [32..127, 160..255] Then
                 Write (Chr(Buffer^[16*i+j]))  { druckbare Zeichen
                                                 ausgeben }
               Else
                 Write ('.');                  { sonst Punkt }
             Writeln
           End;
       End;
     Free_Mem (Long(Buffer), SizeOf(BufferTyp))
   End;
```

Das Schreiben geht dann ganz analog mit den Funktionen "CMD_WRITE" oder "ETD_WRITE".

Außerdem gibt es noch folgende Operationen:

CMD_UPDATE und ETD_UPDATE

Die internen Diskpuffer (im RAM) werden auf Diskette geschrieben. Auch hier haben Sie die Möglichkeit ("ETD_..."), die Operation nur durchzuführen, wenn die Diskette nicht gewechselt wurde.

CMD_CLEAR und ETD_CLEAR

Der Diskettentreiber hält eine gewisse Anzahl von zuletzt benutzten Blöcken im RAM, wodurch z. T. Diskettenzugriffe gespart werden. Mit den beiden "CLEAR"-Funktionen können Sie den Inhalt dieser Puffer als ungültig markieren, evtl. in Abhängigkeit von einem Diskettenwechsel.

TD_FORMAT

Mindestens ein Track der Disk wird formatiert. Dabei zeigen "io_Data" auf Daten (mindestens 1 Track = 11*512 Bytes lang), mit denen die Spuren gefüllt werden sollen, "io_Offset" muß auf einen Spuranfang zeigen (Spurnummer * 11 * 512) und "io_Length" gibt die zu formatierende Länge in Bytes an, muß aber ebenfalls ganze Tracks umfassen (io_Length:= Spuranzahl * 11 * 512).

Hier haben Sie nicht die Möglichkeit, das Ganze nur bei ungewechselter Diskette durchzuführen. Es wird kein "Verify" der formatierten Spuren durchgeführt.

TD_SEEK

Bewegt den Kopf zu einer bestimmten Position.

Zu guter letzt seien hier noch die wichtigsten Fehlernmmern aufgeführt, die das Trackdisk-Device zurückgibt:

21	Sektor-Header nicht gefunden
24	Prüfsummenfehler in Headerfeld
25	Prüfsummenfehler in Datenfeld
26	Falsche Anzahl von Sektoren auf einem Track
27	Sektor-Header unlesbar
28	Diskette schreibgeschützt
29	Diskette wurde gewechselt
30	Fehler bei SEEK
31	Nicht genug Speicher frei
32	Falsche Gerätenummer (z. B. Laufwerk nicht angeschlossen)
33	Falscher Laufwerkstyp
34	Drive wird schon benutzt

Weiterführende Literatur

Natürlich erhebt diese Einführung in die AMIGA-Systemprogrammierung keinen Anspruch auf Vollständigkeit. Eine Übersicht über alle Möglichkeiten des Betriebssystems und der Hardware würde den Rahmen dieses Handbuchs sprengen, und so bleibt mir nichts anderes übrig, als Sie auf weiterführende Literatur zu verweisen.

Dabei ergibt sich das Problem, daß PASCAL auf dem AMIGA bisher ein Schattendasein fristet (vielleicht kann KICK-PASCAL das ändern?). Deshalb beziehen sich die meisten Bücher über die Programmierung des AMIGA auf C oder Assembler, auf keinen Fall aber auf PASCAL. Wenn Sie sich in einer dieser beiden Sprachen auskennen, dürfte es nicht weiter schwierig sein, passende Literatur zu finden.

Als erstes wären die "AMIGA Reference Manuals" von Addison-Wessley zu nennen, die aus nicht weniger als vier Bänden bestehen (und jeweils ca. 80 DM kosten). Sie richten sich an alle, die es ganz genau wissen wollen. Achtung, bisher nur in Englisch erschienen!

Das "Amiga Intern" von Data Becker geht recht ausführlich auf die Hardware, Exec und das DOS ein und endet mit einem Kapitel über Devices.

Wer Literatur zur Sprache Pascal möchte, muß auf allgemeine Pascal-Bücher zurückgreifen. Da wäre z.B. die "Einführung in Pascal" von Rodnay Zaks, erschienen im Sybex-Verlag, das die elementaren Pascal-Eigenschaften behandelt und sich zum Erlernen der Sprache eignet.

Komplizierter geht es zu bei "Professionelles Turbo Pascal" von Herbert Schildt. Dieses Buch, erschienen bei McGraw-Hill, behandelt beispielsweise Sortier- und Suchroutinen, Listen und Bäume, dynamische Zuweisungen und Programme aus Mathematik und Statistik. Die vorgestellten Programme sind ohne große Probleme mit KICK-Pascal lauffähig.

ANHANG

A) Compiler-Fehlermeldungen

"," expected.
":" expected.
"(" expected.
")" expected.
"." expected.
".." expected.
"=" expected.
"[" expected.
"]" expected.
":=" expected.
"OF" expected.
"BEGIN" expected.
"END" expected.

Der Compiler vermutet, daß an der angegebenen Stelle das entsprechende Symbol stehen müßte. Dabei ist aber zu beachten, daß er den Fehler nur grob analysiert und im Grunde genommen absolut keine Ahnung hat, was Sie an dieser Stelle eigentlich vorhatten. Deshalb kann der eigentliche Fehler auch an einer ganz anderen Stelle liegen.

Absolute variables must get declared single.

Werden Variablen als "ABSOLUTE" deklariert, müssen sie einzeln deklariert werden. Falsch wäre z. B.

```
VAR a,b: integer ABSOLUTE Adr;
```

richtig dagegen

```
VAR a: integer ABSOLUTE Adr;
    b: integer ABSOLUTE Adr;
```

Boolean Expression expected.

Der Compiler erwartet hier einen Boole'schen Ausdruck, for example als Bedingung bei IF..Then, While..Do oder Repeat..Until.

Branch too far.

Sprünge, die inhärent bei PASCAL-Strukturen wie IF, REPEAT oder CASE auftreten, dürfen im Code nur über eine Strecke von maximal 32767 Bytes reichen. In der Praxis dürfte dieser Fehler kaum auftreten, denn das bedeutet, daß sich z. B. eine Schleifenstruktur über tausende von Programmzeilen erstreckt. Falls Sie so etwas tatsächlich einmal zustande bringen, sollten Sie den Umfang der Struktur

durch Zerlegen in Prozeduren verringern - denn Unterprogrammaufrufe (und auch GOTO-Sprünge) sind von dieser Regelung ausgenommen.

Can not jump into a structure.

Man kann mit GOTO nicht von außen z. B. in eine FOR-Schleife hineinspringen.

Can not open mathieeedoubbas.library!

Vor dem ersten Compiler-Lauf wird die genannte Bibliothek geöffnet. Sie muß sich also im "Libs:"-Verzeichnis befinden.

Can not write unit.

Die Interface-Datei eines Units konnte nicht geschrieben werden.

Can't copy this file.

Bei einem Include-Dateien mit zwei angegebenen Pfaden konnte die Datei zwar über den zweiten Pfad gelesen, nicht aber an den ersten Pfad kopiert werden.

Can't create directories.

Beim Kopieren von Includefiles und Units werden bei Bedarf Unterverzeichnisse automatisch erstellt. Das ging hier schief.

CASE needs an ordinal expression.

Mit CASE können nur geordnete Datentypen verglichen werden, also beispielsweise keine Real- oder String-Werte.

Compiler stack overflow.

Der Task-Stack war nicht groß genug für den Compiler (dürfte kaum vorkommen).

Constant expected.

Tritt an verschiedenen Stellen auf, u. a. im CONST-Definitionsteil und in der CASE-Anweisung. Hier darf nur eine Konstante stehen.

DO expected.

Hinter WHILE, WITH oder FOR fehlt das zugehörige DO. Vielleicht haben Sie sich aber auch in der Bedingung vertan, so daß der Compiler deren Ende an dieser Stelle vorzeitig vermutet.

Double definition

Ein Label wurde zweimal gesetzt.

Double use of Identifier.

Sie haben in einem Deklarationsteil (z.B. Variablendeklaration) einen Bezeichner definiert, der auf selber Ebene schon einmal deklariert ist.

Element type is too large.

Sets dürfen maximal 65536 Elemente enthalten, wobei die Grenzen auf Vielfache von 16 auf- bzw. abgerundet werden.

Error in string

Beim Einschließen von Steuercodes in Stringkonstanten haben Sie irgendetwas falsch gemacht.

Error while reading unit.

Dateifehler beim Lesen des Interface-Teils eines Units

Field identifier expected.

Dies ist kein gültiger Feldbezeichner dieses Recordtyps.

File expected.

Hier hat eine Variable eines Dateityps zu stehen.

File is not of required Type

Die mit "{$link ... }" oder als Unit eingeladene Datei ist keine Objektdatei.

FILE OF FILE is not allowed.

Der Elementtyp eines File-Typs darf weder selbst ein File-Typ sein noch einen solchen enthalten.

File not found

Eine Include-Datei konnte nicht gefunden werden, vielleicht, weil sie gar nicht existiert.

Filename expected.

Hinter der Compiler-Directives "incl" und "path" müssen jeweils Dateinamen stehen, die mit einfachen ' oder doppelten " Hochkomma einzuschliessen sind.

Filename too long.

Die maximale Länge von Dateinamen für Include-Dateien beträgt einschließlich Pfad 80 Zeichen.

Files must be VAR parameters.

Wenn sie eine Datei als Parameter an eine Prozedur oder Funktion übergeben wollen, muß dies in Form eines VAR-Parameters geschehen.

First constant is higher than second one.

Bei der Definition eines Ausschnittstyps (z. B. eines Array-Indextyps) haben Sie den Bereich so gewählt, daß die obere Grenze kleiner als die untere ist.

FOR needs an ordinal type.

Die Schleifenvariable einer FOR-Schleife muß von einem geordneten Datentyp sein (insbesondere nicht REAL).

Functions can return simple, pointer or string types only.

Der Datentyp, den eine Function zurückgibt, kann sein:

- Ganzzahl, Boolean, Char oder ein Ausschnitt eines solchen Typs
- Real
- Pointertyp
- String

Garbage at end of program.

Hinter dem Programmende steht noch etwas.

Global procedure without parameters expected.

Prozeduren, die als Parameter bei "AddExitServer" angegeben werden, müssen global sein und dürfen keine Parameterliste haben.

Identifier expected.

Der Compiler meint, daß hier ein Bezeichner stehen müßte.

Identifier not defined in that translation unit

Ein Bezeichner wurde falsch qualifiziert, z. B. "System.Hallo", so daß er im entsprechenden Unit nicht gefunden werden konnte.

Illegal type for textfile input

Daten dieses Typs können nicht mit Read/Readln aus einer Textdatei (zum Beispiel von der Tastatur) gelesen werden.

Illegal assignment or compiler failure Type conflict or compiler failure Double panic - general protection!

Hier haben Sie den Compiler ertappt. Es ist intern etwas eingetreten, was eigentlich nicht sein darf. Zu Deutsch: Es handelt sich um einen Bug des Compilers oder um einen fälschlich nicht gemeldeten Fehler Ihres Programms. Wahrscheinlich liegt es daran, daß er sich in der Vielzahl der kombinierbaren unterschiedlichen Datentypen verheddert hat.

Illegal type for textfile output.

Daten dieses Typs können nicht mit Write/Writeln in eine Textdatei (z. B. auf den Bildschirm) geschrieben werden.

IMPLEMENTATION expected.

Im Interfaceteil eines Units trat ein Fehler auf.

Incompatible parameterlists.

Die Parameterliste (oder auch, bei Funktionen, der Ergebnistyp) eines Prozedur/Funktion-Parameters paßt nicht.

Integer expression expected.

An dieser Stelle muß ein ganzzahliger Ausdruck stehen, z. B. als Längenangabe von String[..] oder Feldbreitenangabe bei Write(-Ln).

Label expected.

Hinter dem "LABEL"-Symbol kann entweder ein gewöhnlicher Bezeichner oder eine Ziffernfolge (ganze Zahl) stehen.
oder: Was hier hinter GOTO steht, ist nicht als Label deklariert worden.

Label on wrong level.

Ein Label darf nur auf der Ebene gesetzt werden, auf der es deklariert wurde, insbesondere nicht in einem Unterprogramm dieser Ebene.

Lib-base must be of LongInt or Pointer type.

Die Basisadresse einer Library muß ein Ausdruck eines der genannten Typen sein (vielleicht liegt es daran, daß Sie "IntBase", "DosBase" oder "SysBase" deklariert und dann ein System-Include-Datei geladen haben).

Magic number does not match.

Ein Unit wurde zwar gefunden, aber der Compiler stellte fest, daß es gar kein Unit ist.

Memory overflow

Während des Compilierens ist der Arbeitsspeicher übergelaufen. Verlassen Sie KICK-Pascal und versuchen Sie es mit größerem Arbeitsspeicher noch einmal.

Name too long (max. 80 chars)

Dateinamen von Units u. ä. dürfen einschließlich Pfad höchstens 80 Zeichen lang werden.

Nesting overflow

Die Schachtelungstiefe von Prozeduren und Funktionen darf höchstens 16 betragen.

No free memory for buffer.

Aus Geschwindigkeitsgründen wird für Include-Dateien jeweils ein Pufferspeicher angelegt. Dieser Error sagt, daß dafür kein Speicher mehr reserviert werden konnte.

No free memory for BSS-Hunk.

Für die als STATIC oder EXPORT deklarierten Variablen ist kein Speicher mehr frei.

No free memory for relocation table.

Der Compiler muß gelegentlich für die Relokationstabelle (was das ist, braucht Sie nicht unbedingt zu interessieren, falls Sie es nicht wissen) zusätzlichen Speicher reservieren. Wenn Sie diese Meldung erhalten, haben Sie ihr RAM so vollgeknallt, daß nicht mehr einmal ein paar KByte für diese Tabelle frei waren.

No free memory for unit.

Es war kein Speicher zum Einlinken einer Datei frei.

Numeric expression expected.

Hier muß ein numerischer Ausdruck stehen.

Only global Procedures/Functions can get exported.

Lokale Prozeduren und Funktionen dürfen nicht mit "EXPORT" ausgeführt werden.

Ordinal Type expected.
Ordinal expression expected.

Hier dürfen nur geordnete Datentypen stehen, also Boolean, Char, Ganzzahl- und Aufzählungstypen.

Offsets must be integer.

Die Adressoffsets in einer Library müssen natürlich ganzzahlig sein.

Out of range

Zur Compile-Zeit wurde bereits festgestellt, daß irgendwo ein Bereichsüberlauf stattfindet, z. B. bei "chr(257)".

Overflow at nesting of Incl-files

Include-Dateien dürfen maximal 25-fach ineinander verschachtelt werden. Auch die Interface-Teile von Units werden beim Lesen intern als Include-Dateien behandelt (was sie aber nicht sind), so daß dieser Fehler auch hier auftreten kann.

Overflow in floating-point-arithmetic.

Bereits zur Compile-Zeit trat bei Fließkommarechnungen ein Überlauf auf, z. B. durch eine zu große Konstante.

Please set an unit directory.

Wenn Sie Units mit "USES" einbinden oder ein Unit compilieren (wobei es automatisch abgespeichert wird), müssen Sie im Pull-Down-Menü "Suchpfade" mindestens ein Directory dafür angeben. Am besten setzen Sie mit dem PascalPrefs-Programm einen entsprechenden Eintrag in die Config-Datei.

Please use "Round" or "Trunc".

Gleitpunktausdrücke können nicht über normale Typkonvertierungen wie z. B. "Long(x)" oder Integer(Sqrt (a))" in ganze Zahlen gewandelt werden. Dazu sind die genannten Funktionen zu verwenden.

'PROCEDURE' or 'FUNCTION' expected.

Dieser Fehler tritt nur innerhalb einer Library-Deklaration auf.

"Program" expected.

Am Anfang muß nicht unbedingt "Program" stehen. Dieser Fehler wird immer gemeldet, wenn das Programm nicht sinnvoll anfängt.

Short Circuit of Units.

Ein Units ruft sich selbst auf, evtl. auch indirekt über weitere Units.

Simple type expected.

(wird u. a. bei EXCHANGE gemeldet): hier sind nur Variablen einfacher Datentypen erlaubt: Geordnet, Real oder Pointer.

String exceeds line.

Eine Zeichenkette darf das Zeilenende nicht überschreiten. Vielleicht haben Sie bei einem String das Hochkomma am Ende vergessen oder eins gesetzt, wo Sie gar keines setzen wollten.

String expression expected.

Falscher Parameter bei "Insert", "Delete" o. ä.

String too long.

Die maximale Länge von Stringkonstanten beträgt 400 Bytes.

Syntax error in compiler directive.

In einer Compiler-Directive haben Sie irgend etwas falsch gemacht.

Temporary strings can not get assigned to "STR"-pointers.

Die String-Verknüpfung "+" sowie die Funktionen "Concat", "Copy" und "RealStr" liefern einen temporären String zurück, der keinen dauerhaften (also nur "temporären") Speicherplatz besitzt. Deshalb kann man diese Funktionsergebnisse zwar an STRING-Variablen zuweisen, nicht aber an STR-Variablen.

Dieser Fehler wird auch gemeldet, wenn ein temporärer String als Parameter des Typs "Str" übergeben wird.

THEN expected.

Hinter einem IF fehlt das zugehörige THEN. Vielleicht haben Sie sich aber auch in der Bedingung vertan, so daß der Compiler deren Ende an dieser Stelle vorzeitig vermutet.

These types can not be converted.

Eine derartige Typkonvertierung ist nicht möglich.

This function identifier can not be a statement.

Am Anfang eines Befehls - oder dessen, was der Compiler dafür hält - steht unerlaubterweise ein Funktionsname.

This Procedure/Function can not be used as parameter.

Nicht jede Standardfunktion kann als Parameter übergeben werden. Beispiele: chr, sin, cos oder srqt dürfen, ord, eof oder sqr dagegen nicht. Faustregel: Eine Funktion, bei der Parameterliste und Rückgabetyp immer gleich sind, darf übergeben werden ("sqrt" bildet immer von Real auf Real ab, "Sqr" aber von jedem numerischen Typ auf denselben, besitzt also mehrere Signaturen). Für Prozeduren gilt natürlich sinngemäß das selbe.

TO expected.

Hinter einem FOR fehlt das zugehörige TO oder DOWNTO. Vielleicht haben Sie sich aber auch im Ausdruck vertan, so daß der Compiler deren Ende an dieser Stelle vorzeitig vermutet.

Type expected.

Hier muß ein Typ stehen, egal, ob als Typbezeichner oder als Beschreibung.

Type conflict.

Zwei Datentypen passen nicht zueinander, z.B. die beiden Konstanten, die die Grenzen eines Ausschnittstyps festlegen.

Type conflict of operands.

Die Typen zweier Operanden passen nicht zusammen.

Type-free pointers must not be used here.

Bei NEW und DISPOSE dürfen Sie keine Variablen des Typs "Ptr" als Parameter angeben, da der Compiler ja nicht weiß, wie viel Speicher er dafür reservieren muß.

Type identifier expected.

Hier darf nur ein Typbezeichner stehen. Typbeschreibungen sind nicht zulässig.

Unable to open main file.

Nach dem Übersetzen eines Units konnte die angegebene Hauptdatei nicht gefunden werden.

Undefined Type.

Dieser Datentyp wurde zwar schon erwähnt, aber noch nicht definiert.
Beispiel: "Type a=^b; c=b;" ergibt diesen Fehler.

Unexpected end of source.

Der Quelltext war "überraschend" (für den Compiler) zu Ende.
Mögliche Ursache: Sie haben einen Kommentar begonnen und die Klammer am Ende vergessen.

Unit not found.

Das angegebene Unit wurde nicht gefunden.

Unknown identifier or Syntax error.

Diese Fehlermeldung kann so gut wie alles bedeuten und kann auch an so ziemlich jeder Programmstelle auftreten. Entweder wurde ein Bezeichner nicht deklariert oder ein grober Fehler in der Syntax gefunden.

USES expected.

Hinter "FROM <Projektname>" muß "USES" stehen.

Variable at odd address

Einer als ABSOLUTE deklarierte Variable, die mehr als ein Byte belegt, wurde eine ungerade Adresse zugewiesen. Hier spielt der Prozessor nicht mit.

Variable expected.

Hier soll eine Variable stehen.

WITH expects a variable of a record type.

Hinter WITH muß eine Record-Variable stehen (mit 99%-iger Wahrscheinlichkeit haben Sie an Stelle einer Record-Variablen einen Pointer auf eine solche gesetzt).

Wrong index type.

Sie haben wahrscheinlich beim Array-Zugriff einen Index mit einem falschen Datentyp verwendet.

B) Laufzeitfehler

Subscript out of range.

Der Index eines Arrays (wozu natürlich auch die Strings gehören) war außerhalb des angegebenen Bereichs.

Diese Fehlermeldung erhalten Sie nur, wenn im Menü "Optionen/Compiler" der Punkt "Indexbereich" abgehakt ist.

Out of range.

Einem Ausschnittstyp wurde ein Wert außerhalb des deklarierten Bereichs zugewiesen.

Eine solche Bereichsüberprüfung wird nur vorgenommen, wenn die Compiler-Option "Subrange testen" während des Compilierens angewählt ist.

Division by zero.

Es wurde mit "/", "div" oder "mod" durch 0 dividiert.

Error: Can't open xxx.library

a) Beim Befehl OpenLib wurde die angegebene Bibliothek nicht gefunden.
b) Die "mathtrans.library", die für verschiedene Real-Funktionen benötigt wird, befindet sich nicht im Verzeichnis "libs:".

Out of memory.

Das freie RAM ist knapp, so daß "New" oder "Alloc_mem" nicht ausgeführt werden konnte.

Freemem failed.

a) Mit DISPOSE wurde versucht, ein dynamisches Objekt zu löschen, das nicht existiert bzw. bereits gelöscht wurde.
b) Es wurde versucht, mit Free_Mem Speicher freizugeben, der nicht mit dem Befehl Alloc_mem reserviert wurde (wenn Speicher mit der Exec-Funktion "AllocMem" angefordert wurde, kann er nur mit "FreeMem", NICHT mit dem Befehl "Free_Mem" freigegeben werden).

User Break.

Sie haben das Programm mit der Taste F10 abgebrochen.

Error in Case.

In einem CASE-Statement ohne ELSE-Zweig trat eine nicht vorgesehene Alternative auf (also ein Wert, der nicht als Konstante angegeben ist).

Error in CloseLib

Die angegebene Library existiert nicht, wurde bereits geschlossen oder war nicht mit OpenLib (sondern z. B. mit der Exec-Funktion "OpenLibrary") geöffnet worden.

Overflow error.

Es trat in einer arithmetischen Operation (z. B. Integer-Addition) ein Überlauf auf. Um eine solche Fehlermeldung zu erhalten, müssen Sie vor dem Compilieren die Compileroption "Arithm. Überl." aktivieren.

Intuition error

Ein Window oder Screen konnte nicht geöffnet oder geschlossen werden.

Exec error

Bei einer Exec-Funktion lief etwas schief.

C) Demoprogramme

AnalysatorII - das Analysis-Tool

Dieses leistungsfähige Programm ist in drei Module aufgeteilt:

- AnaII.p: Das Hauptprogramm einschließlich aller Grafikroutinen
- AnaIO.p: Ein- und Ausgabeprozeduren, darunter ein Zeileneditor mit History funktion
- Analysis.p: Prozeduren und Funktionen zum Parsen, Auswerten und Differenzieren von Formeln.

Die Bedienung erfolgt ausschließlich über die Tastatur.

Analysator II kann bis zu 300 Funktionen verwalten. Sie heißen f0 bis f99, g0 bis g99 und h0 bis h99. Dabei ist "f" identisch mit "f0" (g und h entsprechend). Die Ableitung einer Funktion erhält man, indem man hinter ihren Namen einen Apostroph (bzw. mehrere) setzt.

Funktionen sind stets von der Variablen "x" abhängig. Außerdem gibt es 400 Variablen: "a" bis "d", ebenfalls mit optionaler Nummer von 0 bis 99. Analosator kennt auch die Konstanten "e" und "pi" und alle gängigen mathematischen Funktionen, z. B. sin. arccos, sqr (Quadrat), ln, sqrt (Quadratwurzel), sgn (Vorzeichen), usw.

Das Multiplikations-"*" kann da, wo es in der Mathematik auch üblich ist, weggelassen werden (z. B. 2sin(5x), 3pi/2). Vorsicht bei "e": Hinter einer Zahl wird dies als Zeichen für den Exponenten gehalten, d. h. 1.2e+4 ist 1200 und nicht 1.2*e+4.

Die Befehle:

funktionsname "=" ausdruck
(Definiert eine der 300 Funktionen.)

variablenname "=" ausdruck
(Weist einer Variablen einen Wert zu.)

"?" ausdruck (Gibt einen Funktionsterm oder den Wert eines numerischen Ausdrucks aus.)

"t" ausdruck (Druckt eine Wertetabelle. Die Grenzen und die Schrittweite werden abgefragt. Auch dabei sind als Eingaben beliebige numerische Ausdrücke erlaubt.)

"l+", "l-" (Schaltet den Druckmodus ein und aus. Dabei werden die Ausgaben der Befehle "?" und "t" auf den Drucker kopiert.)

"l=" dateiname (Leitet die Druckerausgabe von "l+" in eine andere Datei um.)

"p" ausdruck, ausdruck, ...
(Die Graphen der angegebenen Funktionen werden gezeichnen. Das Zeichnen kann mit einer beliebigen Taste vorzeitig abgebrochen werden.)

"r" (Der Bereich, in dem "p" malen soll, wird abgefragt. Bei Programmstart gültiger Default ist: -10 < x < 10, -5 < y < 5.)

"q" (Quit, Programmende)

Eingabebeispiel:

```
f=sin(sqr x)   { Definiert Funktion f=sin(x^2) }
?f''           { zweite Ableitung ausgeben lassen }
f3 = f'''      { dritte Abl. berechnen und an f3 zuweisen }
?f3(2)         { an der Stelle 2 auswerten }
?17+4          { Ganz einfach diesen Ausdruck berechnen }
g=f^2          { g=sin(x^2)^2 }
h=g(2-5x)      { h=sin((2-5x)^2)^2 }
?h'(5)         { erste Abl. berechnen und an Stelle 5 auswerten }
t 0.1*f'(2x)   { Wertetabelle dieser Formel drucken. }
r              { Koordinatenbereich festlegen }
p f,f',f''     { Funktion f und ihre beiden ersten Ableitungen im zuvor angegebenen
                 Bereich zeichnen lassen }
```

berno.p - Bernoullizahlen

Berechnet die Bernoullizahlen, und zwar nicht als Fließkommazahl (was ja trivial wäre), sondern als Bruch. Da das Programm aber nur mit 16-Bit-Integers rechnet, kommt es ab B(7) durch Überlauf zu falschen Ergebnissen und bei B(11) gerät das Programm sogar in eine Endlosschleife, sofern Sie nicht die Compiler-Option "Arithmetik-Überlauf" angewählt haben. Die LongInt-Version "berno2.p" kommt schon etwas weiter.

cal.p - Kalender

Mit diesem kleinen Programm können Sie Wochentage berechnen und Kalender ausdrucken. Dem Programm (nur vom CLI startbar) ist als Parameter ein Datum zu übergeben. Dabei gibt es vier Formate:

cal <Jahr>	Jahreskalender
cal <Monat> <Jahr>	Monatskalender
cal <Tag> <Monat> <Jahr>	Wochentag berechnen
cal <Tag> <Monat> <Jahr>	Differenz zum heutigen Datum berechnen

Für die letzte Option muß die Systemzeit korrekt gesetzt sein. Einmal davon abgesehen, daß das europäische Format "Tag-Monat-Jahr" verlangt wird, ist das Programm recht tolerant: Monate können als Nummer angegeben werden, wie in

```
cal 26. 7. 1931
```

oder als String:

```
cal 21. März 90
```

Monatsnamen können englisch oder deutsch angegeben werden, wobei immer nur die ersten drei Buchstaben beachtet werden, z. B.

```
cal 18-Feb-1990
```

Es ist des weiteren egal, ob man die Parameter mit einem Punkt, Komma, Strich oder einfach nur einem Leerzeichen trennt. Zulässige Jahreszahlen sind 1600 bis 9999; wird ein Jahr von 0 bis 99 angegeben, bezieht sich das auf das 20. Jahrhundert. Man kann also kaum etwas falsch machen.

directory.p - Diskinhalt ausgeben

Dieses Programm heißt zwar wie "Dir", ähnelt aber eher dem CLI-Befehl "List". Davon einmal abgesehen, demonstriert es die Benutzung der Dos.library und die Auswertung von Parametern.

IFF.p - ILBM-Bilder laden und zeigen

Dieses Programm kennen Sie vielleicht schon von der Version 1.0 von KICK-PASCAL. Es ist aber inzwischen stark verbessert worden. Zum einen konnte die Ladegeschwindigkeit auf ein recht gutes Niveau gesteigert werden, so daß man das lauffähige Programm wirklich sinnvoll benutzen kann. Außerdem kann man es jetzt auch direkt von der Workbench starten, indem man es entweder als "Default Tool" im Icon eines Bildes einträgt oder zuerst das Bild-Icon und dann mit "geSHIFTetem" Doppelklick ein Exe-File von "IFF.p" startet.

Beim Start aus dem CLI kann man jetzt den Dateinamen als Parameter angeben, etwa

```
Iff Bild
```

Es gibt eine ganze Menge Optionen, die beim Start aus dem CLI gewählt werden können, indem man ihre Kennbuchstaben mit einem vorangestellten "-" in beliebiger Reihenfolge vor oder hinter dem Dateinamen angibt, etwa:

```
IFF Bild -n -t
```

Die Optionen kann man auch beim Workbench-Start benutzen, indem man im Icon des Bildes als "Tool Type" ein bestimmtes Wort schreibt.

Hier ist eine Liste aller Optionen:

- **-n** Das Bild wird auf keinen Fall im Interlace-Modus dargestellt (ist besser für die Augen). Als Tool Type ist "NoInter" oder "NoLace" zu wählen.
- **-i** Das Bild WIRD im Interlace-Modus ausgegeben. Tool Type: "INTER" oder "LACE".
- **-h** erzwingt Hold-And-Modify-Modus. Tool Type: "HAM". Diese Option ist weitgehend überflüssig, denn bei Lo-Res-Bildern mit 6 Planes wird defaultmäßig der HAM-Mode gewählt.
- **-l** Option für Lo-Res-Modus (320*256 Punkte). Tool Type: "LoRes".
- **-m** Schaltet auf hohe Auflösung (640 Punkte breit). Tool Type: "HiRes" oder "MedRes".
- **-x** Wählt den Extra-Halfbrite-Modus, was aber nur bei Lo-Res-Bildern mit 6 Bitplanes geht. Beim Start von der Workbench ist "XHALF" als Tool Type anzugeben.
- **-t** "Talk"-Option: Während des Ladens werden Informationen über das Bild und die Bilddaten ausgegeben. Diese Option ist nur beim CLI-Start anwählbar.

Das Bild wird so lange angezeigt, bis eine der Tasten Space, Return, Enter oder Escape gedrückt wird. Wenn das Bild größer als die Bildschirmauflösung ist, z. B. weil ein Hi-Res-Interlace-Bild mit der Option "-l" geladen wurde, kann der angezeigte Bildausschnitt mit den Cursortasten gescrollt werden.

Die Bilddaten werden in der vorliegenden Version nicht mehr direkt auf den Bildschirm geladen, sondern zuerst in einen Puffer abgelegt, von wo jeweils der gerade aktuelle Ausschnitt (bei übergroßen Bildern) mit Zuhilfenahme des Blitters auf den Screen kopiert wird.

Dieses Programm ist nicht nur ungemein nützlich - welcher Amiganer hat nicht im Laufe der Zeit eine mehr oder weniger umfangreiche IFF-Bildersammlung angelegt und will sie des öfteren ansehen, ohne erst ein Malprogramm o. Ä. zu laden - sondern auch lehrreich: es demonstriert folgende immer wiederkehrende Programmieraufgaben:

- Parameterübernahme vom CLI und der Workbench. Wie Sie sehen können, ist es überhaupt nicht schwierig, einen Dateinamen von der Workbench zu holen. Man muß nur die Include-Datei "workbench/startup.h" einlesen und ein wenig mit Zeigern und Records hantieren.
- Schnelle Dateihandhabung: Die alte Version von "IFF.p" war allein deshalb so lahm, weil die Daten in kleinen Portionen - Byte für Byte, teilweise auch Langwortweise - aus der Bilddatei gelesen wurden. Dabei macht sich dann bemerkbar, daß das DOS des Amiga nicht gerade schnell ist. In der neuen Version legt "IFF.p" einen 5 KByte großen Pufferspeicher an, aus dem dann byteweise gelesen werden kann, bis er leer ist und neu geladen werden muß.
- Benutzung des Blitters. Aus einem Pufferspeicher, in den die Bilddaten zunächst geladen werden, werden sie mit dem Blitter blit(t)zschnell in die Bitmap des Screens kopiert.

integra.p - Integrationsverfahren

Dieses Programm demonstriert Procedure/Function-Parameter.

lister.p - Textdateien

Kleine demo für die Dateihandhabung von KICK-PASCAL. Das Programm gibt eine Textdatei mit Zeilennummern auf den Drucker oder in eine andere Datei aus.

Mon.p - Systemmonitor

Mit diesem Programm können Sie die Innereien des AMIGA-Betriebssystems erkunden (allerdings bisher nur in eingeschränkter Form). Es ist ein Monitorprogramm, das auch Strukturen des AMIGA-Betriebssystems kennt.

Am Anfang wird die ExecBase-Struktur dargestellt. Mit den Cursortasten können Sie die einzelnen Felder mit dem Cursor durchlaufen und mit der Taste "J" sowohl (bei mit "*" gekennzeichneten Feldern in eine Unterstruktur als auch bei Feldern des "L"-Typs) zu einer anderen Struktur springen. Die jeweils 20 letzten Sprünge werden intern gespeichert, so daß Sie mit der Taste "B" jeweils wieder zurückspringen können.

Um dieses Programm compilieren zu können, benötigen sie ca. 150 KByte reservierten Arbeitsspeicher. Nachher werden Sie aber feststellen, daß der Speicher mit Quelltext und Code zusammen nicht einmal halb voll ist. Wie kommt's? Nun, mit "incl" werden hier einige Dateien eingeladen, die jeweils wieder mehrere andere Dateien ein-"includen".

Nun reicht es natürlich nicht, wenn der Compiler sich diese Dateien (zusammen ca. 40 KByte) einfach nur durchliest - er muß sich auch merken, was 'drinsteht. Aus Geschwindigkeitsgründen wurde hier keine dynamische Speicherverwaltung implementiert; vielmehr werden die Daten im Arbeitsspeicher abgelegt. Daher kann es dazu kommen, daß der Compiler irgendwann einen Speicherüberlauf meldet.

rescue.p - Quelltextretter

Dies ist ein ausgesprochen nützliches Programm: Hiermit können Sie Ihre KICK-PASCAL-Quelltexte "retten", wenn ihnen der Rechner abgestürzt ist oder wenn Sie das PASCAL-System versehentlich verlassen haben, ohne ihr Programm vorher abzuSAVEn.

Zur Funktionsweise: KICK-PASCAL legt unmittelbar vor der Stelle, an der Anfangs- und Endadresse des Sourcetexts stehen, eine Kennung ab, die wie folgt aussieht:

- String "KICKPASCAL", abgeschlossen mit 2 Nullbytes
- Integerzahlen 26731 und 4711
- Anfangs- und Endadresse des Texts, jeweils LongInteger.

Das Programm "Rescue" durchsucht nun den gesamten Speicher nach der Kennung und stellt so fest, wo der gesuchte Text liegt. Findet es den Text, gibt es Speicherbereich und Textlänge aus und fragt nach dem Dateinamen, unter dem der Text gesichert werden soll. Danach wird weitergesucht, denn es ist nicht sicher, ob dies der richtige Text war, da es möglicherweise mehrere solcher Kennungen gibt.

Zur Anwendung: Wenn Sie lediglich einen "Software Error - Task held"- Requester bekommen haben, besteht in der Regel kein Problem. Schwieriger wird es, wenn bereits nach einer Guru Meditation neu gebootet wurde oder Sie KICK-PASCAL versehentlich verlassen haben, denn dann stehen die gesuchten Daten völlig ungeschützt im Speicher! Für diesen Fall sollten Sie ein fertig compiliertes Objectfile von "Rescue" zur Verfügung haben und es sofort (!) anwenden (ggf. sogar beim Booten die Startup-Sequence sicherheitshalber vorzeitig abbrechen).

Das Programm geht davon aus, daß das FastRam, falls vorhanden, bei Adresse $c00000 beginnt. Sollte dies bei ihnen nicht zutreffen, können Sie diesen Wert (am Anfang des Quelltextes) entsprechend ändern.

Seit Version 2.0 von KICK-PASCAL enthält das Rescue-Tool eine in Assembler geschriebene Unterroutine. Sie können das Programm aber auch neu compilieren, wenn Sie keinen Assembler besitzen, denn der Assemblerteil liegt als fertig einlinkbare Objektdatei "Rescue.l" vor. Den Assembler-Quelltext finden Sie auf ihrer Diskette gleich zweimal: Im KICK-ASS-Format ("Rescue.c") und als ASCII-Textdatei ("Rescue.s").

sieb.p - Primzahlberechnung/Setdemo

Berechnet nach dem bekannten Algorithmus die Primzahlen bis zu einer wählbaren Obergrenze.

D) Kurzübersicht

1.) Wort-Symbole (reserviert Pascal-Schlüsselwörter):

AND	ARRAY	BEGIN	CASE
CONST	DIV	DO	DOWNTO
ELSE	END	FILE	FOR
FUNCTION	GOTO	IF	IN
LABEL	MOD	NIL	NOT
OF	OR	PROCEDURE	PROGRAM
PACKED	RECORD	REPEAT	SET
SHL	SHR	THEN	TO
TYPE	UNTIL	VAR	WHILE
WITH	XOR		

Diese Liste entspricht (außer SHL, SHR und XOR) dem Jensen-Wirth-Standard.

2.) Direktiven:

ABSOLUTE	EXPORT	EXTERNAL	FORWARD
FROM	IMPLEMENTATION	IMPORT	INTERFACE
LIBRARY	MODULE	OTHERWISE	STATIC STRING
UNIT	USES		

Der Jensen-Wirth- (ISO 7185-) Standard kennt nur eine Direktive, nämlich FORWARD (die Direktives in diesem Sinn sind nicht zu verwechseln mit den Compiler-Direktiven wie z.B. {$incl...}!) Direktiven sehen zwar auf den ersten Blick wie Wortsymbole aus, werden aber compilerintern wie Bezeichner (Identifier) behandelt.

Dadurch können sie z.B. vom Programmierer "überdefiniert" werden, d. h. Sie können eine Variable "Forward" oder einen Datantypen "String" nennen. Letzteres wird in der Tat auch häufig in Standard-Pascal-Programmen getan ("TYPE String=packed Array[1..xxx] of Char" ist eine sehr beliebte Deklaration). Aus diesem Grunde wurden auch die KICK-PASCAL-typischen Schlüsselwörter wie "STRING" oder "LIBRARY" in Form von Direktiven realisiert.

3.) Konstanten

```
true
false
MaxInt          = 32767
MaxLong         = 2147483647
MaxLongInt      = 2147483647
MaxCard         = 65535
MaxWord         = 65535
MaxShortCard    = 255
MaxByte         = 255
MaxShort        = 127
MaxShortInt     = 127
Pi              = 3.14159265358979
```

4.) Typbezeichner

a) Boolean = (false,true)

b) Ganzzahltypen:
   Integer = -32768 .. 32767
   ShortInt = Short = -128 .. 127
   LongInt = Long = -2147483648 .. 2147483647
   ShortCard = Byte = 0 .. 255
   Cardinal = Word = 0 .. 65535

c) Char = chr(0) .. chr(255)

d) Real

e) Text = FILE OF Char

f) Str

g) Ptr

5.) Variablen

Input : text
Output : text
Sysbase: ptr Absolute 4
DosBase: ptr
MathBase: ptr
MathTransBase: ptr
FromWB: boolean
ParameterStr : str
ParameterLen : integer
StartupMessage: ptr
Mem : Array[0..MaxLongInt] Of Byte Absolute 0

6.) Prozeduren

AddExitServer (proc)
Assign (datei,name)
BlockRead (datei, variable, anzahl)
BlockWrite (datei, variable, anzahl)
Buffer (datei, bytes)
CBreak
ClrScr
ClrEol
Close (datei)
CloseConsole (devpointer)
CloseLib (libpointer)
Close_Screen (screenhandle)
Close_Window (windowhandle)
Dec (ordinalvar)
Delay (secs*50)
Delete (stringvar, pos, len)
DelLine
Dispose (pointer)
DisposeAll
Error (string)
Exchange (var1,var2)
Free_Mem (adresse,grösse)
Get (datei)
GotoXY (x,y)
Inc (ordinalvar)
Insert (string,stringvar,pos)
InsLine New (pointer)
OpenLib (pointervar, name, version)

Page [(datei)]
Put (datei)
Randomize
Read (readparameterlist)
ReadLN (readparameterlist)
Reply_Msg (msgpointer)
Reset (datei)
ReWrite (datei)
SetStdIO (consoleptr)
Val (string, numvar, intvar)
Write (writeparameterlist)
WriteLN (writeparameterlist)
WriteCon (devpointer, string)

7.) Funktionen

Abs (numerisch): numerisch
Addr (var): LongInt
Alloc_Mem (größe,flags): LongInt
ArcTan (x): Real
Break (n): Boolean
Chr (int): Char
Concat (str1, str2, ...): String
Copy (string, pos, len): LongInt
Cos (x): Real
Eof [(datei)]: Boolean
EoLN [(datei)]: Boolean
Exp (x): Real
FreeStack: LongInt
Get_Msg (port): Ptr
Hi (i): Integer
IntStr (long): String
Length(string): LongInt
Ln (x): Real
Lo (i): Integer
Odd (int): Boolean
OpenConsole (window): Ptr
Open_Screen (x,y,breite höhe,tiefe,hfarb,vfarb,mode,name): Ptr
Open_Window (x,y,breite,höhe,farbe,IDCMP,flags,name,
screen,minw,minh,maxw,maxh): Ptr
Ord (ordinal): Integer
Pos (str1, str2): LongInt
Pred (ordinal): ordinal
Pwr10 (int): Real

Random: Real
Random (i): Integer
ReadCon (devpointer): Char
RealStr (r,i): String
Round (x): LongInt
Sin (x): Real
SizeOf (typ): LongInt
Sqr (numerisch): numerisch
Sqrt (x): Real
StrLen (string): Integer
Succ (ordinal): ordinal
Swap (i): Integer
Trunc (x): Real
Upcase (ch): Char
Wait (maske) : Long
Wait_Port (port): Ptr

INDEX

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

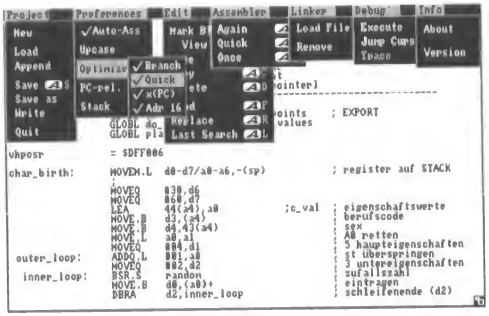
W

X

Z

KICK-ASS

Der besondere Assembler



KICK-ASS ist mehr als ein Assembler. Durch seinen integrierten **EDITOR**, **MONITOR**, **DEBUGGER**, **TRACER** und **LINKER** ist KICK-ASS ein leistungsstarkes Werkzeug, das die Entwicklung von Assembler-Programmen einfach, schnell und bequem macht.
KICK-ASS bietet sowohl dem **Einsteiger** als auch dem **Profi** eine ideale Entwicklungsumgebung. So wurde z.B. KICK-Pascal komplett mit KICK-ASS entwickelt.

Die Features

- blitzschnelle **Direkt**-Assemblierung während der Eingabe
- vielfältige **Optimierungen** (einzeln abschaltbar)
- automatische **Syntaxüberprüfung** bei jeder Befehlseingabe
- **Fullscreen-Editor** mit vielen komfortablen Befehlen
- übersichtliche **Befehlsformatierung**
- Verwendung von einfachen **Befehlsabkürzungen**
- **integrierter LINKER** (extern linkfähig mit BLINK)
- **DEBUGGER** zur komfortablen Programmanalyse
- komfortabler **TRACER**
- **Sektionierung** (DATA-, BSS- und CHIP-Hunks)
- **SEKA-befehlskompatibel**
- erzeugt sehr **kompakten Kode**
- komplett **menü- und tastengesteuert**
- deutsches Handbuch

Optimieren und Debuggen

Einer der wichtigesten Punkte bei einem Assembler ist die Erzeugung schnellen Codes. Beim KICK-ASS können hierzu mehrere Optimierungen eingestellt werden. **QUICK** wandlt "move", "add" und "sub" in MOVEQ, ADDQ und SUBQ. **BRANCH** erzeugt möglichst kurze Sprünge. **ADR16** ersetzt absolute 32-Bit- durch 16-Bit-, **X(PC)** dagegen in PC-relative Adressen.

Zur Fehleranalyse dienst der Debugger mit seinem komfortablen Trace-Modus. Programmabläufe können damit schrittweise mitverfolgt und Fehler schnell gefunden werden. Während des Tracens werden alle Register sowie die System-Flags und deren Manipulation angezeigt. Auftretende Fehler wie z.B. "privileg violations" werden abgefangen.
Der Tracer ist somit ein wichtiges Werkzeug bei der Fehlersuche.

Unverbindliche Preisempfehlung **DM 89.-**

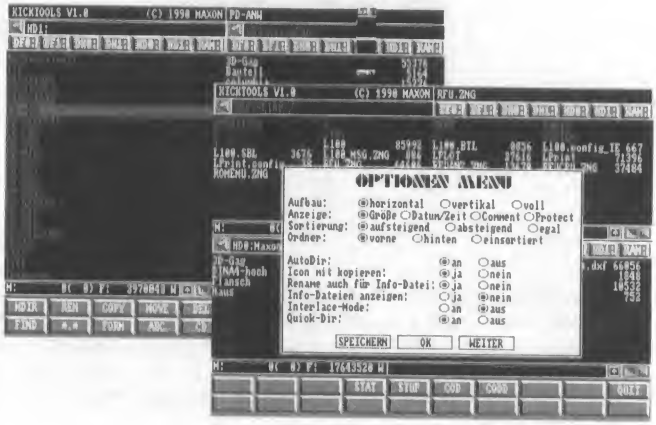
MAXON
computer gmbh

KICK-TOOLS

Die professionelle Arbeitshilfe

IM ÜBERBLICK

- **KOMFORTABLE CLI-BEFEHLE** (MOVE, FIND, PRT, TYPE, PIC, SHOW u.a.)
- **Shortcuts** und benutzerdefinierte Funktionstasten
- Flexible Darstellung der Verzeichnisse (horizontal, vertikal, bis zu 220 Dateien)
- **AutoDir**- und **QuickDir**-Funktion
- Vielfältig **konfigurierbar** (Aufbau, Anzeige, Sortierung, Ordner, Icons, Devices usw.)
- **Iconify**-Funktion
- **Neuartige Linkviren-Kontrolle** (mit Dateilänge und Prüfsumme)
- **Schnelles Backup-Programm**
- **Datei- und Diskettenkodierung**



Komfortabele CLI-Befehle

KICKTOOLS ist ein Hilfsprogramm, das dem AMIGA-Anwender ein **Höchstmaß an Komfort** bei der täglichen Arbeit mit dem AMIGA bietet. An erster Stelle steht hier der Umgang mit Dateien und Verzeichnissen. Das CLI und auch die Workbench bieten hier nur wenig und deshalb gestalten sich solche Arbeiten oft umständlich und zeitaufwendig. KICKTOOLS bietet hier Funktionen, die **weit über das Übliche** hinausgehen z.B. eine **AutoDir**-Funktion, die das Verzeichnis jeder Diskette sofort einließt und dabei einen schnellen **QuickDir**-Algorithmus verwendet. Dateien können auf die übliche Weise untereinander dargestellt werden, empfehlenswert ist jedoch die **horizontale Darstellungsweise**, wie sie sich auf MS DOS-Rechnern schon lange durchgesetzt hat. Dies hat den Vorteil, daß fast doppelt so viel Dateien auf einmal dargestellt werden können. Einzelne Dateien oder ganze Verzeichnisse können komfortabel mit der Maus ausgewählt werden und danach stehen Ihnen **vielfältige Funktionen** wie z.B. Kopieren, Verschieben (MOVE), Löschen usw. zur Verfügung. Es ist aber auch sehr einfach, sich einen **Text** oder ein **IFF-Bild** anzeigen oder einen **Sound** vorspielen zu lassen. Wichtig dabei ist, daß Sie die KICKTOOLS-Oberfläche nicht verlassen müssen um solche Aktionen durchzuführen. Sehr angenehm sind Funktionen wie **FIND** zum Suchen einer bestimmten Datei auf einer Festplatte oder **PRT** bzw. **PRTD** zum Ausdrucken einer Datei bzw. des Verzeichnisbaums. Auch Programme können **direkt** von KICKTOOLS aus gestartet werden. Außerdem können Sie die von Ihnen **häufig benutzten** Programme auf eine separate Knopfleiste legen und dann über **Shortcuts** aufrufen.

Datei/Diskettenkodierung

Schützen Sie Ihre Daten vor fremden Zugriffen, durch eine Verschlüsselung mit einem Passwort, das nur Ihnen bekannt ist. Sie haben mit KICKTOOLS die Möglichkeit, einzelne Dateien, aber auch ganze Disketten zu Verschlüsseln.

Neuartige Viren-Kontrolle

Linkviren sind sehr unberechenbar und Ihre Erscheinungsform sehr unterschiedlich. Sie hängen sich an Programme an und werden erst zu bestimmten Zeiten oder Aufgrund bestimmter Ereignisse aktiviert. Der einzige zuverlässige Schutz ist eine regelmäßige Kontrolle des Datenbestands. Mit KICKTOOLS können Sie alle Dateien Ihrer Festplatte auf veränderliche Dateilängen und Veränderung des Inhalts (Prüfsumme) überprüfen. Nur so können Sie neue Linkviren aufspüren, bevor Sie großen Schaden anrichten.

Schnelles HD-Backup

Nicht nur Viren, sondern auch der übliche Verschleiß oder unerwartet auftretende Defekte der Festplatte können Ihre wichtigen Daten in Sekunden zunichte machen. Deshalb ist ein regelmäßiges Backup, zumindest der wichtigsten Verzichnisse Ihrer Festplatte unverzichtbar. KICKTOOLS bietet Ihnen hierzu eine einfache Lösung in Form eines integrierten Programmteils an. Vorteile sind:

- Eigenes Diskettenformat (über 900KB/Diskette)
- Datenverschlüsselung (optional)
- Unabhängige Datendisketten (bei einer defekten Diskette ist der Rest des Backups nicht gefährdet)

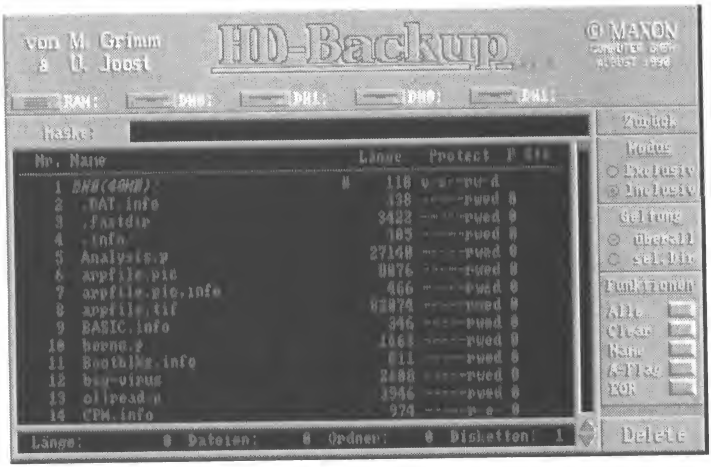
Unverbindliche Preisempfehlung: **DM 99.-**

MAXON computer gmbh

MAXON Computer GmbH
Schwalbacher Straße 52
D-6236 Eschborn

HD BACKUP

Festplatten-Backup



Es ist sicher ein gutes Gefühl, alle Daten auf einer Festplatte zu haben, doch nicht nur Viren, sondern auch Verschleiß oder plötzlich auftretende Fehler können wertvolle Daten in Sekundenschnelle zerstören. Eine regelmäßige Sicherung der gesamten Festplatte oder zumindest der wichtigsten Datenbestände ist daher unverzichtbar.
An ein Datensicherungsprogramm werden die verschiedensten Anforderungen gestellt. HDBackup bietet viele Funktionen und Optionen an, um Ihnen die Arbeit beim gezielten Sichern Ihrer Daten zu erleichtern. Die Oberfläche würde nach diesen Gesichtspunkten gestaltet, so daß sich alle Funktionen mit der Maus auswählen lassen. Ein Auswahlmenü hilft bei der Selektion, wenn man nur einen Teil der Daten sichern will. So können z.B. alle Quelltexte einer Partition mit *.c ausgewählt werden. Sie können aber auch einfach nur alle Dateien mit nicht gesetztem Archiv-Flag sichern lassen. Die Anzahl der notwendigen Disketten wird sofort berechnet. Die Verify-Option und die Möglichkeit, die Backup-Disketten jederzeit zu Überprüfen, gewährleisten ein hohes Maß an Datensicherheit.

Unverbindliche Preisempfehlung: **DM 99.-**

Daten

- grafische Benutzeroberfläche
- volle Mausunterstützung
- komfortable Dateiauswahl mit Optionen (Pattern, Archiv-Flag)
- unterstützt bis zu 4 Laufwerke bei Backup/Restore
- Verify-Option und nachträgliche Backup-Kontrolle
- leistungsfähige Packalgorithmen (bis zu 1.6 MB Diskette)
- eigenes Diskettenformat (über 900 kB/Diskette)
- Protokoll auf Bildschirm, Drucker, Datei
- deutsche Benutzerführung und Anleitung

MAXON computer gmbh
MAXON Computer GmbH
Schwalbacher Straße 52
D-6236 Eschborn

MaxonCAD

Die neue Dimension in Design und Konstruktion

Die benutzerfreundliche Oberfläche von MaxonCAD mit dem Fenstermenü (rechts) für schnellen Zugriff auf die wichtigsten Funktionen.

MaxonCAD ist das leistungsstarke CAD-Programm für alle Anwender vom Hobby-Designer bis hin zum professionellen Konstrukteur. Seine transparente und flexible Gestaltung macht es offen für jede Art von Anwendung, von der mechanischen Konstruktion bis hin zur Architektur.

Integriert in eine komfortable Benutzeroberfläche, die durch ihre Geradlinigkeit leicht erlernbares, komfortables und effizientes Arbeiten ermöglicht, kombiniert MaxonCAD die wichtigsten Leistungsmerkmale, die für den Anwender von Bedeutung sind:

- Sehr hohe Arbeitsgeschwindigkeit und schneller Bildaufbau
- Einfachste und schnelle Bedienung des gesamten Programms über Pulldown-Menüs und Dialogboxen
- schneller Zugriff auf die am häufigsten benutzten Funktionen über Shortcuts und eine aussagekräftige Iconleiste
- Darstellung eines Gummibands bei allen Zeichenfunktionen
- Parametereingabe mit der Maus und/oder der Tastatur

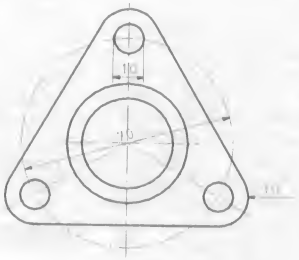
Der außerordentlich große Leistungsumfang von MaxonCAD wird unter anderem durch folgende Features abgesteckt:

- Verschiedene geometrische Grundelemente, wie Linien, Kreise, Kreisbögen, Ellipsen, Ellipsenbögen und Text.
- Vielfältige Konstruktionswege für jedes Grundelement, wie z.B. Parallelen, tangentiale Linien an zwei Elemente, berührende Kreise zu zwei Elementen, tangentiales Ansetzen von Kreisbögen
- Umfassende und leistungsstarke Funktionen zum Nachbearbeiten von Zeichnungen, z.B. Drehen, Skalieren, Spiegeln, Trimmen und Runden
- Vielfältige Punkteingaben, wie z.B. Schnitt-, Lötfuß- und Berührpunkte
- Halbautomatische Bemaßung
- Änderungen in der Bemaßung sind auch nachträglich möglich
- Einbinden von Symbolen
- Erstellung eigener Symbolbibliotheken
- Spline-Interpolation
- Schraffuren mit beliebigem Schraffurwinkel und Schraffurabstand
- Zeichnen auf beliebig vielen Ebenen
- Hochwertiger Ausdruck über alle AMIGA-Druckertreiber
- Plotteranteuerung über HPGL-Dateien
- Lesen und Ausgeben von Zeichnungen im DXF-Format

Gegenüber dem „großen" MaxonCAD ist die Student-Version in folgenden Punkten eingeschränkt:

- Maximales Bearbeiten von 2 statt 4 Zeichnungen gleichzeitig
- Maximal 6 Zeichenebenen
- Maximal 4 Bibliotheken gleichzeitig im Speicher
- Trennen nur mit Trennpunkt
- Nur Trimmen von einem Element
- Nur Runden von zwei Elementen
- Nur Fassen von einem Element
- Keine Plotterausgabe

Damit stellt MaxonCAD Student eine preiswerte Möglichkeit für alle Anwender dar, zu geringen Kosten nahezu die volle Leistungsfähigkeit von MaxonCAD zu nutzen.



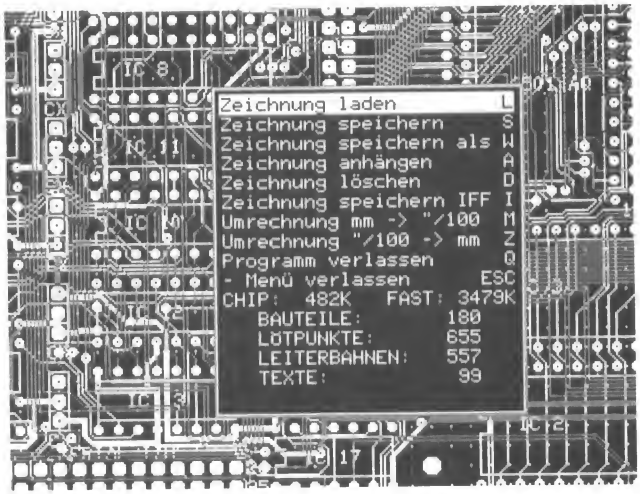
MAXON computer gmbh
Schwalbacher Straße 52
D-6236 Eschborn
Tel.: 06196-481811

Geeignet für alle AMIGA mit mindestens 1 MByte Speicher.

MaxonCAD DM 449.-
MaxonCAD^{Student} DM 249.-
(unverbindlich empfohlener Verkaufspreis)

L100

Platinenlayout-Programm



L100 ist ein leistungsfähiger Platinen-Layout-Editor. Er ist komplett tastaturgesteuert um das Positionieren und Ziehen von Leiterbahnen bei einer Genauigkeit von 1/100 Zoll zu erleichtern. Umfangreiche Funktionen unterstützen die Entwickung auch größerer Layouts bis 257x257 bzw. 346x192 mm. Als Funktionen stehen unter anderem zur Verfügung: Leiterbahn (löschen, restaurieren, kopieren, duplizieren, verdicken), Symbol (zeichnen, definieren, auflösen), Lötpunktgröße, PAD (liegend, stehend), Bauteil (zeichnen, definieren, übernehmen, auflösen), Pinnummern (eingeben, autom.), Text zeichnen (0/90/180/270 Grad), IFF-Speichern. Beim Ausdruck kann zwischen Leiterbahnseite, Bauteilseite, Texte, nur Bauteile und Lötstoppmaske, einem Faktor zwischen 0.5 und 2.0, Mehrfachdruck und globale Leiterbahndicke gewählt werden, wobei der Druck in der jeweils höchsten Auflösung des Druckers erfolgen kann.

Daten

- Die Auflösung des Editors beträgt 1/100 Zoll
- Vier Platinengrößen:
  (LO/MED) 127/257x127mm (512KB)
  (HI) 257x257/346x192mm(1 MB Speicher)
- 7 verschiedene Lötpunkte und PADs
- automatische Pin-Numerierung
- Echtzeit-Lupenfunktion und Übersichtsmodus
- superschnelles Softscrolling
- variable Cursorschrittweite und Rasterfunktion
- optionales Fadenkreuz (senkrecht oder diagonal)
- erweiterbare Bauteilbibliothek
- Das Druckprogramm unterstützt EPSON-kompatible Drucker (24- und 8-Nadler) bis 400x400 DPI.
- HPGL-kompatibles Plot-Programm
- deutsche Benutzerführung und Anleitung

MAXON
computer gmbh

MAXON Computer GmbH
Schwalbacher Straße 52
D-6236 Eschborn

Unverbindliche Preisempfehlung: **DM 99.-**

Face The Music

Der achtstimmige Soundcomposer

FTM

Face The Music (FTM) ist ein sehr leistungsfähiges, achtstimmiges Soundprogramm mit vielfältigen Einsatzmöglichkeiten. Die zugrundeliegende achtstimmige Soundroutine ist voll multitaskingfähig. Erstmals werden alle Eigenschaften der Hardware-Soundkanäle simuliert, d.h. getrennte Lautstärkeeinstellung ist ebenso möglich wie die Verwendung von Samples mit Loop und vielfältigen Klangeffekten. Durch einen Programmierkniff gelang es, die maximale Sample-Abtastrate des Amiga zu nutzen, weshalb im Höhenbereich klanglich keine Abstriche mehr gemacht werden müssen. Die Soundprogrammiersprache S.E.L. erschließt Ihnen zusätzlich völlig neue Klangdimensionen.

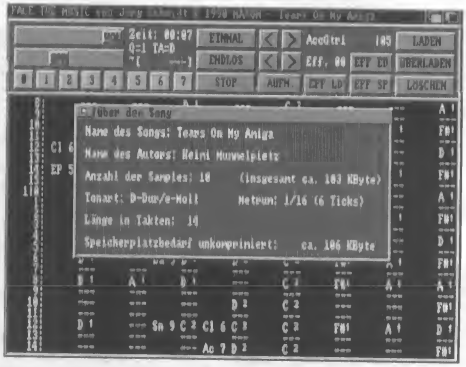
Technische Daten:

- vollständige Emulation von **acht unabhängigen Stimmen** auf dem Amiga (auch Unterstützung von Sample-Loops und kanalmäßig getrennter Lautstärkeeinstellung, Nutzung der vollen Amiga-Samplerate)
- Songgeschwindigkeit beliebig einstellbar (nicht vom Vertical Blank Interrupt abhängig)
- Songs können beliebig lang sein (werden grundsätzlich komprimiert gespeichert) und bis zu **63 IFF- oder Soundtracker**-Samples verwenden
- Samples können in das gesamte FastMem geladen werden
- kein einengendes Pattern-Konzept, stattdessen ein durchgehender Song
- **Loop-Funktion** zur Passagenwiederholung
- **Standard-Spezialeffekte**: Pitchbend ("Ziehen" der Tonhöhe), Volume Down (autom. Ausblenden der Lautstärke)
- freie Wahl der Tonart (das hilft Nicht-Keyboardprofis, leichter Stücke in anderen Tonarten zu schreiben)
- **S.E.L.**, die einzigartige **Sound-Effect-Language (50 Befehle)**, ermöglicht **beliebige Klangmanipulationen**, wie sie bisher nur bei Synthesizern möglich waren. Die Möglichkeiten reichen von der Programmierung komplexer Rhythmen über **Phasing-Effekte** bis zu Ringmodulation von vier LFOs.
- automatische **Akkordgenerierung** durch Noteneingabe zusammen mit Alt-Taste
- keine Beschränkung auf 4/4-Takt (die Taktauflösung kann bis zu einer 1/96-Note eingestellt werden)
- **voll multitaskingfähig**
- läuft auch auf dem **Amiga 3000** und unter **KICK/WB 2.0**
- **deutsche Benutzerführung**

Unverbindliche Preisempfehlung: **DM 99.-**

MAXON computer gmbh

MAXON Computer GmbH
Schwalbacher Straße 52
D-6236 Eschborn
Tel.: 06196 - 48 18 11



Der komfortable Editor ist die Schaltzentrale von FTM

Der Editor:

- superschneller Song-Editor mit komfortabler Benutzeroberfläche
- **übersichtliche Darstellung der Noten** in Textform
- soweit wie möglich Verzicht auf Hexadezimalzahlen, stattdessen „Klartext" (z.B. stets Verwendung des Sample-Namens statt einer laufender Nummer)
- **komfortable Block-Funktionen** (Ausschneiden, Kopieren, Tonhöhe und Lautstärke manipulieren, vertikal und horizontal umkehren usw.).
- speicherbare Voreinstellungen, z.B. Suchpfade für Soundverzeichnisse

Eingabe:

- Noteneingabe oder freies Improvisieren (in **Echtzeit**) über Tastatur oder **MIDI-Keyboard**
- Drei-Oktaven-Klaviatur
- Berücksichtigung der Tonart bei der Eingabe möglich
- Berücksichtigung von **MIDI-Keyvelocity**

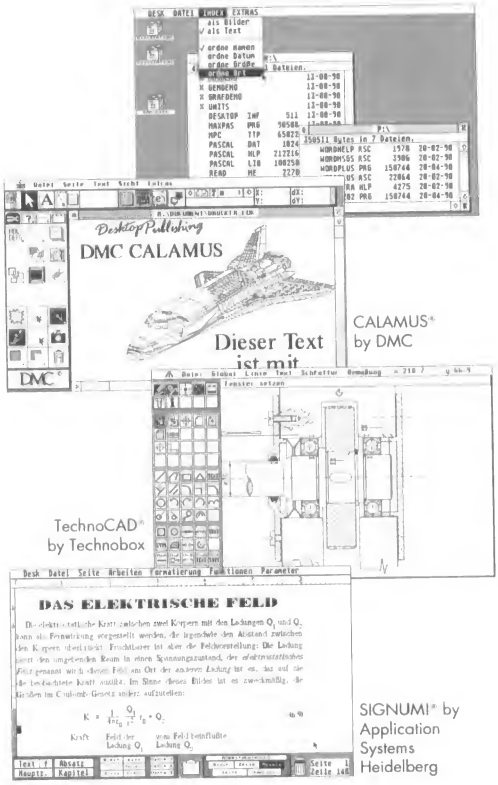
Lieferumfang:

- FTM inkl. einer Sammlung gesampelter Sounds und Instrumente
- ausführliches **deutsches Handbuch**
- **Abspielroutine zur Einbindung in eigene Programme**, die das direkte Abspielen komprimierter Song-Module erlaubt
- **multitaskingfähige, achtstimmige Abspielroutine** für CLI und Workbench

CHAMÄLEON

ATARI ST-Emulator

CHAMÄLEON - der ATARI ST-Emulator - ermöglicht es, mit Ihrem AMIGA in die große Welt der ST-Software vorzudringen.



CALAMUS® by DMC

TechnoCAD® by Technobox

SIGNUM!® by Application Systems Heidelberg

- Für alle AMIGA 500, 1000, 2000
- Unterstützt alle Auflösungen des ATARI ST
- Parallelport-Emulation
- Direktes Lesen und Schreiben von ST-Disketten
- Nahezu volle ST-Geschwindigkeit
- Benötigt Original ATARI-TOS
- Ein Patch-Programm sorgt dafür, daß auch kritische Programme, die auf absolute Adressen zugreifen, auf dem AMIGA laufen (z.B. SIGNUM!).

Optionales Modul zur Aufnahme der ST-ROMs erhältlich!

MAXON
computer gmbh

GASAL

Die fexible Präsentationssprache

GASAL ist das ideale Programm für alle Anlässe, bei denen Daten und Grafiken **ansprechend präsentiert** werden sollen. Durch eine BASIC-ähnliche Script-Sprache mit über **60 Befehlen** für die Bereiche **Grafik**, **Sound**, **Animation**, **Effekte** und **Programmlogik** ist es auch ungeübten Anwendern möglich, eine individuelle und überzeugende **Präsentation** zu erstellen. GASAL kann bis zu 10 Bildern, 50 BOBs, 8 Sprites, 10 Sounds und 1 Soundtracker-Song gleichzeitig verwalten. Auch **interaktive Abfragen** der Maustasten und Mauskoordinaten sind möglich. Damit ist der Ablauf einer Präsentation vom Benutzer beeinflußbar, wodurch sich weitere Anwendungsgebiete (**Schulung**, **Multiple-Choice-Test**, **Quiz** usw.) erschließen. Eine Vielzahl von Funktionen sind schon als leicht aufrufbare Befehle vorhanden, GASAL ist aber weitaus **flexibler** und gestattet die einfache Erstellung eigener **effektvoller Überblendeffekte**. Aus den fertigen Script-Dateien kann ein selbstablaufendes Programm erzeugt werden, das **frei weitergegeben** werden darf.

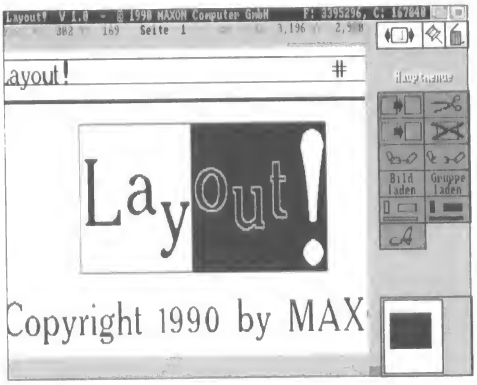
Daten

- Geeignet für: **Präsentationen aller Art, Schaufensterwerbung, Vorführungen, Vorträge, Schulungen, Ton-Diashows**
- Verwaltet **10 Bilder, 50 Bobs** und **10 Sounds** gleichzeitig.
- **Einfache Steuerung** mit einer Script-Sprache
- **Über 60 Befehle** für Text, Grafik, Sound und Animation
- Fertige **Überblend-Effekte**
- **Flexible Gestaltung eigener Effekte**
- **Interaktive Abfragen**
- **Superschnell** durch vollständige Assembler-Programmierung
- Erzeugt **eigenständig** ablauffähige Programme
- Mit Beispielen, Bildern, Bobs, Sprites, Sounds
- Deutsches Handbuch

Unverbindliche Preisempfehlung **DM 99.-**

LAYOUT!

Grafische Gestaltung am Amiga



Gestalten Sie Ihre **Grußkarten, Urkunden, Poster, Spruchbänder, Einladungen, Visitenkarten, Briefpapier und vieles mehr**. Layout! stellt Ihen dazu alle notwendigen Funktionen zur Verfügung. Sie können **beliebige IFF-Bilder** einlesen und in **Graustufen** (Helligkeit, Kontrast, Gamma-Korrektur, Konturen glätten) konvertieren lassen. Vergrößern, Verzerren, Spiegeln, einen Bereich herausschneiden, Invertieren und andere Funktionen lassen Ihnen alle Freiheiten bei der Bearbeitung der Grafiken. Wenn Sie alles zu Ihrer Zufriedenheit bearbeitet und positioniert haben, dann kommt der Text an die Reihe. Verwendet werden kann **jeder AMIGA-Font** in beliebiger Größe und natürlich lassen sich auch hier die Funktionen anwenden, die für Grafiken gelten. Bei der Wahl der Blattgröße sind Sie ebenfalls kaum eingeschränkt, denn Layout! verwaltet bis zu **30.000x30.000 Pixel**. Auf diesem riesigen Papier können Sie **blitzschnell scrollen**, denn es wird das Prinzip der **Superbitmap-Technik** verwendet. Der Ausdruck bringt dann endlich die Arbeit zu Papier. Hierbei zeigt Layout! was in ihm steckt, denn sie können Ihrem Drucker das Letzte entlocken - überzeugen Sie sich selbst ...

Daten

- **Riesige Arbeitsfläche** (bis zu 30.000x30.000 Punkte)
- **Superschnelles Scrolling** (durch Superbitmap-Technik)
- Beliebige **AMIGA-Fonts**
- Verarbeitung von **IFF-Bildern**
- **Graustufenkonvertierung** mit umfangreichen Optionen
- Gruppenbildung
- Schnelle **Preview**-Funktion
- Hilfen: **Gitter, Blatteinteilung, Koordinatenkreuz**
- Ausdruck in der **maximalen Auflösung** des Druckers
- Deutsche Benutzerführung und Anleitung

Unverbindliche Preisempfehlung **DM 59.-**